Separation of functional and time interference concerns for efficient AMC 20-193 compliance

Damien Chabrol, Jean Guyomarc'H, Fabien Siron, Guillaume Phavorin, Sam Thompson, Eric Jenn, François Thurieau

Separation of functional and time interference concerns for efficient AMC 20-193 compliance

Damien CHABROL[1], Jean GUYOMARC'H[1], Fabien SIRON[1], Guillaume PHAVORIN[1,3], Sam THOMPSON[2], Eric JENN[3], François THURIEAU[4]

[1] Asterios Technologies, Massy, France
{firstname.lastname}@asterios-technologies.com

[2] Rapita Systems Ltd, York, UK
sthompson@rapitasystems.com

[3] IRT Saint Exupéry, Toulouse, France
{firstname.lastname}@irt-saintexupery.com

[4] Safran Electronics & Defense, Massy, France
francois.thurieau@safrangroup.com

***Abstract*—Safety-critical real-time systems must comply with stringent certification requirements, including temporal ones. Failure to comply with these temporal requirements may contribute to system failure. Therefore, timing considerations, such as response times, are of the foremost importance for such systems. As the use of multi-/many-core hardware platforms is becoming inevitable in the avionics industry due to the increasing computing performance required by modern embedded systems, integration activities are getting more and more complex. Increasing concurrency and parallelism exacerbates integration issues and introduces new challenging problems. To answer those challenges, certification authorities have issued guidelines, referenced as A(M)C 20-193, describing some additional objectives to fulfill for multi-/many-core integration. The present paper describes how a time-aware approach, based on the Synchronous Logical Execution Time paradigm (sLET), makes the design and integration of A(M)C 20-193 compliant safety-critical multi-/many-core systems easier by separating functional and time interference concerns.**

Keywords—safety-critical real-time systems, strong determinism, synchronous Logical Execution Time, multicore timing analysis, AMC 20-193.

I. INTRODUCTION

Avionics systems, and more particularly safety-critical ones, are usually subjected to stringent certification constraints to ensure their compliance with safety requirements. Indeed, failure of such systems may result in catastrophic consequences. In particular, for high-criticality real-time systems, functions should be ensured to complete within strict timing constraints. In addition, the avionics industry increasingly relies on modular systems, where multiple applications of possibly different criticality levels can safely share a common hardware platform. Integration activities must ensure that all hosted applications still meet their functional and temporal requirements after their composition in the final system.

Driven by the increasing computing performance required by modern embedded systems and the obsolescence of high-performance single-core processors (SCPs), the avionics industry is moving towards the use of multi-/many-core processor (MCP) hardware platforms. But the increasing parallelism and potential throughput brought by MCPs comes at a cost: integration activities are getting more and more complex, and thus time-consuming and costly [1].

These works have been supported by the French Defense Procurement Agency (DGA) and the French National Research Agency (ANR) in the context of respectively the ASTERLINK and ARCHEOCS projects.

The use of MCPs exacerbates integration issues already present for SCPs. Moreover, it introduces new challenging problems [2]. In particular, when multiple cores are used and different threads of execution simultaneously access the same hardware resource (e.g., a shared memory, a bus, etc.), the hardware must arbitrate these concurrent accesses, effectively introducing additional latency on some of the accesses. This time interference may lead to the violation of the system's temporal requirements. It can also result in new or different data or control coupling paths, and thus functional interference causing the system to behave in a non-deterministic way, or possible data corruption [3]. For example, with MCPs, functional modules can be allocated to different CPU cores, which may create inter-core execution dependencies due to inter-task synchronization (e.g., for communication or to prevent race conditions). Thus, one task could prevent another task, allocated to a different core, from running, and thus forestall any other computation. This may have a significant impact on efficiency and testability for MCP systems.

As per DO-178C/ED-12C, safety-critical systems are usually associated with the highest Design Assurance Levels (i.e., DAL-A or DAL-B). Certification of MCP systems to the highest criticality levels presents the greatest challenge for the avionics industry. As functional and time interference may degrade the system safety, applicants must elaborate an argumentative strategy defending that their systems are indeed robust to such interference. Certification authorities have been working on guidelines to address this specific topic, with the AMC 20-193 document recently issued by EASA [3], and its AC counterpart from the FAA [4]. Few safety-critical MCP systems have been certified until now, and very often at the cost of underusing the additional CPU cores. Thus, new methodologies and tools are needed to support the development and integration process of MCP systems and meet the objectives defined in the A(M)C 20-193.

In this paper, (i) we propose a time-aware strategy based on the Synchronous Logical Execution Time (sLET) paradigm, which encompasses time from system-level design to integration on the final hardware (Section III), (ii) then we show how sLET helps to tackle the MCP functional and temporal interference problem (Section IV), and (iii) eventually, we discuss the application of such time-aware strategy to avionics safety-critical systems (Section V) and illustrate it on an industrial case study, using the combination of **ASTERIOS** and Rapita's on-target analysis tools and **MACH**[178] multicore certification solution (Section VI).

II. POSITIONING

A. Time-aware approaches

Many programming abstractions have been developed to model and reason about real-time systems. For safety-critical systems, they are often coupled with a time-triggered execution model due to its determinism and predictability [5].

The Synchronous-Reactive (SR) model, implemented by synchronous languages [6], totally abstracts execution time to focus on logical instants on which computations are triggered. Each computation should be completed before the next possible instant, and its output must be available before any other computations could use it on the same instant. Thus, SR languages offer both determinism and concurrency. For an implementation on an actual target, logical instants are then mapped to physical time. Multiple logical clocks can exist in SR programs to design multi-rate systems. But due to the causality between computations, induced by instantaneous communications, compilation of SR languages can be quite complex, in particular for multicore platforms (on which computations can be parallelized) [7]. The PRELUDE architecture design language offers a solution to implement multi-periodic synchronous systems, by adding real-time primitives to specify the durations of tasks [8]. Then, the program can be automatically translated into a set of real-time tasks, with periods of tasks computed using clock calculus [9]. Finally, those tasks can be scheduled on-line using policies such as Deadline-Monotonic or Earliest-Deadline-First [10].

The Logical Execution Time (LET) paradigm, implemented for example in the GIOTTO [11] and TDL [12] languages, describes the logical duration taken by computation. Each computation must fit in a logical interval, called LET interval [13]. Furthermore, communications are only made on the boundaries of LET intervals, to ensure determinism. Thus, compared to the SR model, the LET paradigm allows more time variability due to the specified logical duration, which makes concurrent implementations easier. But this comes at the price of a lesser temporal and functional expressiveness, as (i) LET applications in GIOTTO and TDL are limited to strictly periodic tasks and (ii) contrary to SR, LET builds on a delayed communication model.

B. Multicore timing analysis

Multicore timing analysis usually aims to determine safe WCET estimates for software hosted on multicore processors. Different methods can be used: 1) measurement-based analysis, 2) static analysis (deterministic or probabilistic), or 3) hybrid approaches combining both previous points [14]. In every case, the primary challenge that must be overcome is that of multicore interference channels.

1) Interference channels

An interference channel is defined in A(M)C 20-193 as being *'a platform property that may cause interference between software applications or tasks'*. Interference channels can be discovered in many parts of a processor, and are often (but not always) associated with shared hardware resources. Interference channels may be one of the following:

- A bandwidth constraint: e.g., a shared interconnect will typically have a finite bandwidth available which must be shared between any bus masters.
- A space constraint: e.g., shared caches have finite capacity, and tasks that are executing concurrently on different cores may cause evictions of data and instructions that belong to each other, leading to an increase in cache misses and thus execution time.
- An indirect coupling: e.g., a coherency mechanism, whose purpose is to ensure that all levels of cache maintain a consistent view of the state of the memory.

A resource may contain (i) no interference channels, (ii) just one or two, or (iii) a large number (e.g., some complex interconnects, shared caches, and network accelerators can contain more than 10 independent interference channels).

There exist formulations of multicore interference that instead of treating 'interference channels', are built around the concept of 'interference paths'. The pre-eminent example of the latter is the PHYLOG Model Language (PML) [15], which considers all the possible 'initiators' of transactions, all the possible 'targets' for transactions, and all the possible routings between initiators and targets. The assertion follows that if all intersections of these paths are exercised, then all multicore interference will have been assessed. While this approach can provide assurance that all bandwidth constraints are likely to have been tested, additional analysis and test specification may be necessary to ensure that the indirect interference channels have also been adequately covered.

2) Static Analysis and Measurement-based Methods

In older single-core avionic systems, static analysis and simulation can prove useful for timing analysis. However, for complex multicore systems, this is no longer the case. Modern, high-performance processors (particularly multicore ones) have many complex features, such as multilevel caches and DMA engines, which frequently (and, in the case of features like random replacement caches, intentionally) sacrifice temporal determinism in favor of average-case performance. These mechanisms can be very hard to model with sufficient accuracy. Moreover, most modern processors incorporate IP cores from a wide range of sources. So, the silicon vendor may either not be in a position to share (or even build) a complete model of the processor. Furthermore, due to this complexity, errors or inconsistencies in implementation and integration of these IP cores are common, resulting in real-world behavior that doesn't perfectly match the documentation.

Owing to these complexities, A(M)C 20-193 takes a cautious approach to static analysis, and states that *'simulation of those [interference] mechanisms is, therefore, less likely to be representative in terms of functionality or execution time than testing conducted on the target MCP in the intended final configuration, and thus is less likely to detect errors.'* If an airworthiness authority deems that an analysis method is 'less likely to detect errors', then it should generally be avoided.

3) Accounting for pre-emptions

In processors with caches, it is well-documented that pre-emption can be delayed by cache state [16]. In a multicore context, it's also possible for pre-emption to be delayed by operations from other cores, as typically cache coherency transactions will have a higher priority than local accesses.

From a multicore timing perspective, the determination of the maximal pre-emption latency isn't significantly different to the single-core case. However, there may be additional scenarios that need testing (e.g., including cases where other cores are generating many coherency transactions with the intent of maximizing the additional impact).

III. Synchronous Logical Execution Time

Hereafter, we introduce the synchronous Logical Execution Time (sLET) paradigm. This is a generalization of the Psy model introduced during the 90's [17].

A. sLET paradigm

The sLET paradigm combines the benefits (but also some shortcomings) of both the Synchronous-Reactive (SR) and Logical Execution Time (LET) models [18]. It bridges the gap between both approaches by incorporating LET intervals into SR. Thus, it combines SR's properties with more time variability, which makes concurrent implementations (in particular multicore scheduling) easier.

As for the SR model, logical and physical times are considered independent under the sLET paradigm and serve different purposes. Logical time is used to specify the system high-level temporal requirements through an abstraction of time, whereas physical time corresponds to the execution time of the system implementation on a specific hardware platform. As for the SR model, time in sLET is purely logical in the sense that physical time is fully replaced by partial or total ordering between computations. In sLET, logical time is expressed through logical clocks. A logical clock, in the sense described by Lamport [19], abstracts time through a series of events called clock ticks. sLET can be seen as a multiform logical time [20] generalization of LET, hence using multiple logical clocks. A set of clocks C can be constrained by a set of precedence and simultaneity relations (e.g., periodicity).

B. Tasks and Elementary Actions

In sLET, all computations are specified by their activation and termination events, expressed using logical clocks. Computation time intervals can be abstracted by the concept of the *Elementary Action* (EA): an EA is a computation that fits in a sLET interval, bounded by the EA's activation and termination dates. Thus, as depicted in Figure 1, it is defined as a sequence of instructions constrained by two logical dates, referred to as *Temporal Synchronization Points* (TSPs):

- An *activation date* (its earliest start date) defined on some event of a logical clock.
- A *termination date* (its deadline) defined on some other event of a possibly different logical clock (with both clocks related together with a total order).

So, unlike the LET paradigm, sLET does not rely on logical durations: instead, as for SR, it uses logical clock instants (of possibly different non-harmonic clocks) to specify interval boundaries. For example, the EA depicted in Figure 1 has its activation date defined on the second tick of Logical Clock c1 and its termination date defined on the fifth tick of Logical Clock c2. Thus, the sLET interval is defined in terms of clock events and not as a logical duration.

An sLET task is defined by an infinite sequence of EAs. The termination date of an sLET interval corresponds to the activation date of the next interval for the task. Note that it is possible in sLET to have empty logical intervals (i.e., in which no EA from the task can be executed). It allows, for example, to define periodic tasks with constrained deadlines (i.e., with a relative deadline strictly smaller than the period). Moreover, as logical intervals are defined at the EA level (for a same sLET task), this makes it possible to design more complex temporal patterns than strictly periodic tasks.

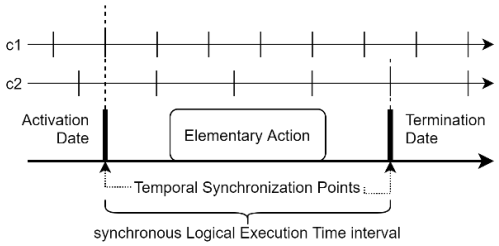

Figure 1: Example of sLET interval (the activation date, respectively termination date, is defined on Clock c1, resp. c2).

C. Visibility principle

As part of a larger system, a task usually consumes and produces data from and to other tasks during its execution. To ensure determinism, sLET inter-task communication is performed through dedicated channels implementing the *Visibility Principle* [21]. Note that by determinism we mean '*the ability to produce a predictable outcome […] based on the preceding operations and data […] in a specific period of time with repeatability*' [3]. Under the Visibility Principle:

- Data produced by an EA over an sLET communication channel will only become visible (i.e., available) to the rest of the system from the end of the EA's interval, i.e., for a logical date greater or equal to the EA's termination date.
- Data can be consumed by an EA from an sLET communication channel only if it has become visible prior to the start of the EA's interval, i.e., for a logical date lesser or equal to the EA's activation date.

The logical date from which the data becomes available to some other EA is referred to as the *Visibility Date*. Usually, this corresponds to the termination date of the EA producing the data, but some sLET communication channels may have their own temporal behavior defined on a different logical clock: in that case, the Visibility Date corresponds to the tick of that logical clock which is greater or equal to the EA's termination date. Note that the producer and consumer's clocks, as well as the one used to define the Visibility Date, must be related with a total order. For example, a data produced by the EA depicted in Figure 1 can only become visible after its termination date (defined on the fifth tick of c2). If we assume that the Visibility Date corresponds exactly to the EA's termination date, this means that another EA can consume this data only if its own activation date is defined on either the same logical tick (fifth tick of c2) or a tick occurring afterwards (e.g., sixth tick of c2, eighth tick of c1, etc.).

So, as for LET, sLET builds on a delayed communication model, which can somehow limit functional expressiveness, in particular compared to SR. Note that, some kind of instantaneous communication can actually be achieved in sLET (as briefly introduced in Section IV), but at the cost of more complex concurrent implementations.

D. Implementation for safety-critical systems

To implement an sLET design on a specific hardware platform, logical time is mapped to physical time. sLET tasks must then be properly scheduled to ensure design timing constraints (i.e., sLET interval bounds). For a single-core platform, the logical ordering of EAs is sufficient to guarantee the correctness of the execution, whereas for MCPs, inter-core synchronization is required to preserve the logical ordering across CPU cores. Moreover, for the resulting tasks'

scheduling to be valid, all EAs must have enough CPU time to complete before the end of their respective intervals. In the remainder of this document, we focus more specifically on static scheduling, based on Time-Division Multiplexing, as it provides strong guarantees on predictability [22] and is generally favored for safety-critical avionics systems. In this case, for deploying an sLET design on a specific hardware target, the user must provide a Time Budget (TB) for each EA, corresponding to the maximum amount of physical time allocated to the computation of that EA. Based on the relations between logical clocks and the provided TBs, a compiler may generate a time-triggered schedule. In that respect, a given scheduling (and so the corresponding sLET implementation) is valid if no EA actually exceeds its allocated TB at run time.

For safety-critical systems, a TB should be an upper-bound on the worst-case execution time (WCET) of the corresponding EA, to ensure that the resulting schedule will always be valid at run time. Thus, safe TB values are synonymous with safe WCET estimates. How such safe TBs could be obtained is discussed in Section V.

IV. TAMING FUNCTIONAL AND TIME INTERFERENCES

Hereafter, we assume that every computation (i.e., EA) is provided with enough physical time (i.e., safe TB) to complete within its logical time constraints.

A. Functional and time interference

As defined in [23], interference corresponds to an alteration of the processor's behavior (e.g., longer delay required for a load operation, etc.) experienced by some part of the software executed on one CPU core, and related to the activity of the remaining software running on the other cores.

As per A(M)C 20-193, time interference can be produced, for example, when the MCP arbitrates simultaneous accesses to shared hardware resources, causing contention for those resources and therefore an increase in execution time [3]. Execution of concurrent software on a different CPU core, and in particular the time interference that may be induced, can result in new or different data or control coupling paths leading to functional interference: a communication buffer may be sometimes read before being written (depending on the producer's and consumer's actual execution times), shared data could be corrupted if accessed in parallel, etc.

B. Preventing functional interference

The sLET Visibility Principle applies the LET communication model [13] to logical clocks. Provided tasks exchange data exclusively through sLET communication channels (H1), their execution is solely driven by their associated logical clocks. It means that communications between tasks become independent from the underlying real-time scheduling (resulting from the implementation of the system on a specific platform). Any schedule complying with the logical constraints defined by the sLET design results in the same functional behavior, as long as physical timing constraints are fulfilled (H2). Thus, if this later hypothesis holds, sLET allows for transparent distribution as functional determinism is ensured whatever the allocation of tasks to CPU cores. This allows dataflow determinism to be achieved. For example, a longer execution of a third-party EA (not involved in the same functional chain) may delay the execution of a producer EA. This could result in the corresponding consumer EA (allocated to another core) being executed beforehand. But, thanks to the Visibility Principle, this has no impact on the dataflow determinism: the consumed data does not depend on the actual execution instants but solely on the sLET intervals bounds. Therefore, functional interference can be prevented by design using the sLET model, as long as tasks exclusively communicate through sLET-based communication mechanisms.

It is the responsibility of the user to ensure that their application complies with both H1 and H2. If H1 is a design constraint, H2 is closely related to the design's implementation on a specific hardware target.

C. Preventing time interference

Time interference between tasks can arise within a single CPU core, e.g., due to cache effects. MCP time interference adds to this 'traditional' time interference, making WCET estimation harder. Here, we focus on MCP time interference, as some extensive work has already been conducted regarding mitigation methods for single-core time interference [14].

Multicore-related time interference can originate from deep and intricate hardware implementation details [24]. Preventing contention (or at least, bounding or minimizing contention) for MCPs reduces potential time interference. A wide spectrum of methods and techniques are available to address this objective, many of which can be used in combination. This paper focuses specifically on temporal exclusion, which can be enabled thanks to the sLET model.

With imperative and non-temporal programming models, concurrent accesses to shared resources (hardware peripheral, software buffer, etc.) are usually guarded, e.g., using mutexes or semaphores. Using sLET, temporal exclusion can be enforced by design, and automatically verified, to prevent such concurrent accesses. Thus, sLET can be used to guarantee that simultaneous accesses to a shared resource never happen, while preventing some issues encountered with mutexes, such as deadlocks. This temporal exclusion is provided through *exclusion groups* [25]. An exclusion group provides a safety property: the EAs it contains must not share any physical date in common. More formally, for a set of sLET tasks T, with $E^t = \{e_i | i \in \mathbb{N}\}$ the infinite sequence of EAs that constitute Task t (e_i being the ith EA of t), an exclusion group G is defined as $G \subset E = \bigcup_t^T E^t : \forall e_i, e_j \in G, e_i \cap e_i = \emptyset$. As a result, a common resource accessed only by EAs from a single exclusion group can only be accessed by at most one EA from that group at a time.

As previously stated, an sLET task consists in an infinite succession of EAs, each bounded by an activation date and a termination date defined on clock events. So, given that all logical clocks can be reduced to a unique global clock, it is possible to define sLET intervals for EAs of a same exclusion group such that they never overlap in time. For example, let us consider the two EAs depicted in Figure 2. Originally (assumably to cope with some high-level timing requirements), EA1's sLET interval is delimited by the first and fourth ticks of c1 and EA2's interval by the first and third ticks of c2. This means that both intervals overlap in time. If EA1 and EA2 need to be part of a same exclusion group, a solution is to modify the sLET design, as depicted in the left sub-figure: EA1's termination date is now defined on the second tick of c1 and EA2's activation date on the second tick of c2. Thus, sLET intervals no longer intersects and temporal exclusion is achieved.

Thus, by constructing exclusion groups, time interferences caused by contention on shared resources may be strongly constrained. However, defining such timing exclusion groups comes at a cost: the user needs to re-design some part(s) of its temporal architecture. This may be arduous work, depending on the temporal patterns of the different tasks. Moreover, this means introducing additional timing constraints (i.e., new or different TSPs) to manage multicore interference (related to a specific integration). This may result in different sLET intervals, which means that overall latencies (derived from high-level requirements) may also change. One solution, described in Section VI, is to deal with timing exclusion only at implementation level: instead of re-designing sLET intervals, scheduling is used to enforce the specified exclusion groups (e.g., by introducing precedence constraints between EAs to avoid concurrent execution). Another solution, using an additional sLET construct, is introduced hereafter.

D. Fractional Temporal Synchronization Points

The sLET paradigm extends 'classic' logical clocks with the concept of *fractional logical clocks*. As any logical clock, a fractional clock abstracts time through a series of clocks ticks, referred to, in this case, as fractional logical ticks. The difference is that a fractional clock is defined with regard to a 'standard' logical clock, such that there is exactly one fractional tick occurring between any two consecutive ticks of the 'parent' logical clock. Note that, as depicted in the right part of Figure 2, this fractional tick can occur anywhere in-between. This means in particular that two fractional clocks defined with regard to the same 'parent' clock cannot be compared as their fractional ticks may occur in any order between two consecutive ticks from the 'parent' clock.

sLET intervals for Elementary Actions can only be defined using 'standard' logical clocks, which means that any EA's activation and termination dates necessarily correspond to logical clock ticks. But an EA execution in its sLET interval can be over-constrained using fractional logical clocks:

- The activation of an EA can be further 'delayed' until after some fractional logical tick, referred to as a *fractional Temporal Synchronization Point*.
- The termination of an EA can be constrained before some other fractional TSP.

For example, as depicted in the right part of Figure 2, EA1 can be constrained to be executed before the second fractional tick of c_f1 (derived from Logical Clock c1) and EA2 after it.

As can be seen in that example, when using fractional TSPs there is no modification of the original sLET interval: the EA's activation and termination dates are left unchanged. As the Visibility Principle is defined with regard to sLET interval boundaries, Visibility Dates for data flows are left unchanged and thus the corresponding latencies. Thus, fractional logical clocks can be used to implement exclusion groups without impeding the original temporal architecture: one EA's execution can be *constrained before* a fractional TSP while the execution of another EA of the same exclusion group is *delayed after* the same fractional TSP, thus ensuring that they don't overlap over time. For example, as depicted in the right part of Figure 2, EA1 and EA2 can no longer be executed concurrently thus enforcing the exclusion group without modifying the original sLET intervals.

Note that, fractional clocks can be used to achieve instantaneous communication in sLET. It is possible to define

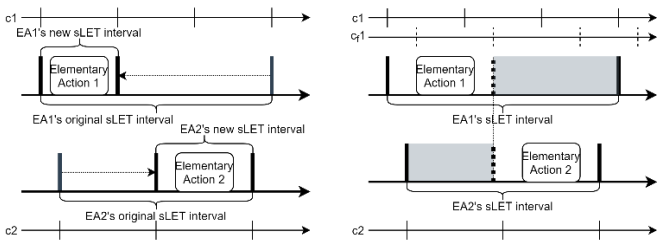

Figure 2: Example of exclusion group between two EAs enforced by: (left) sLET re-design, and (right) using a fractional TSP defined on fractional Logical Clock c_f1 derived from c1 ('classic' clock ticks and TSPs are in plain lines whereas fractional ones are denoted by dashed lines).

sLET communication channels with regard to fractional clocks, instead of 'standard' clocks. In that case, the Visibility Date corresponds to a fractional TSP. On the example from Figure 2, this means that a data produced by EA1 could become visible from the fractional TSP onward, and thus be consumed by EA2.

The additional constraints introduced through fractional TSPs should be ensured by the implementation. Either the corresponding fractional tick is mapped to an actual physical date, as it is the case for 'standard' logical dates, or fractional TSPs are used to derive precedence constraints between EAs' executions that should be ensured by the tasks' scheduling.

V. Towards a Time Budgets evaluation encompassing time interference

As stated in Section III, an sLET implementation relies on compliance with respect to physical timing requirements. In particular, unaccounted time interference defeats this hypothesis. We discuss here how this issue can be addressed.

A. Basis of the approach

As stated before, we focus on static scheduling. A given schedule is valid if no Elementary Action exceeds its allocated Time Budget at run time. To evaluate safe TBs for all EAs, we propose a measurement-based A(M)C 20-193 compliant approach encompassing time interference. Note that other methodologies, as discussed in Section II, are possible.

1) Time Budgets in isolation

The goal of the approach presented hereafter is to compute Time Budgets *in isolation* (from a scheduling point of view, i.e., non-preemptive WCETs [26]). This means that additional delays due to pre-emptions (e.g., additional cache misses resulting from cache evictions caused by the pre-empting tasks) should be accounted for separately when considering the final integration (i.e., with all the application's tasks). As the approach targets multicore integration, this TB in isolation should encompass the maximum possible overhead due to MCP interference. Indeed, dealing at the scheduling level with the interaction between tasks executing in parallel on different CPU cores is much harder than accounting for pre-emptions, and might not always be feasible. In the general case, it's not possible to reason about the test vectors necessary to drive one task to suffer the maximum possible impact from interference caused by another task. Moreover, synchronization is very important for interference impact. Even a single clock cycle of jitter between cores can make a large difference to the interference inflicted on one core by another.

Considering TBs in isolation allows tasks to be handled separately for the timing evaluation, making measurement

campaigns and analyses easier. It also allows for composable approaches and re-usability, and thus possible incremental certification [27], as a single TB could be considered for different multicore integrations of the same task.

2) Incremental approach

The TB evaluation approach presented hereafter is incremental and consists of four main steps:

1. First, an evaluation is performed in single-core to compute a TB upper-bound in isolation, referred to as a *single-core Time Budget*.
2. Then, analyses are conducted to identify possible multicore interference channels and quantify their impact on the different tasks.
3. From those results, multicore interference can be accounted for, either by implementing some mitigation means, or by computing an upper-bound on the maximal overhead to be added to the single-core TB, to derive a *multicore Time Budget*.
4. Finally, multicore TBs for all tasks are verified.

The first three steps are conducted on tasks in isolation. Only the final step is performed on the final configuration.

B. Single-core Time Budget evaluation

First, a TB evaluation is performed in single-core for each task in isolation. As the approach targets TBs in isolation, it means that each task is considered separately, without needing other parts of the application to be present. This is possible as the execution of sLET tasks is solely driven by logical time. So, each task can be executed independently from the others. Of course, inter-task communications, if any, might need to be stubbed. Thanks to the sLET visibility principle, this is easier to achieve as data availability is well-defined.

Measurements are performed using maximizing tests, i.e., exercising the worst-case execution paths for each task at run time, which have to be defined by the applicant on a case-by-case basis. Coverage analyses can be helpful to achieve confidence when building these tests. Moreover, additional metrics might also be collected at this step (e.g., number of memory reads/writes, cache hits/misses, etc.), to (i) construct a profile for the task, which could help understand some software variabilities, and (ii) identify the resources actually used by the task. The high-water mark (HWM), i.e., the highest measured execution time, for each EA can be retrieved from the measurements. Then, a safety margin might be added to get a single-core Time Budget for each EA of the task.

C. Hardware characterization

Hardware characterization deals with the identification and characterization of possible interference channels. To do so, several steps are needed:

1. Hardware resource identification: (i) the resources of the processor need to be identified, and (ii) those that may contain interference channels are singled out.
2. Interference channel identification: any singled-out resource is analyzed in detail, to identify the possible interference channels it contains [23].
3. Interference channel characterization: any non-fully mitigated interference channel is characterized on target to determine its possible effect.

Note that both the hardware resource and interference channel identifications are paper activities and are performed using any available technical documentation and datasheets.

1) Hardware resource identification

Hardware resource identification is required by A(M)C 20-193's MCP_Planning_2 objective. It is important to note at this stage that not all multicore interference channels arise from the explicit sharing of resources. For example, cache coherency mechanisms can cause interference even when only private cache memories are being accessed.

2) Interference channel identification and characterization

Interference channel identification and characterization are partly to satisfy A(M)C 20-193's MCP_Resouce_Usage_3 objective. Characterization can also be used to provide evidence that some interference channels can have no practical or measurable timing impact. This activity should be conducted on target. Interference generators can be used for this purpose [28]. For each channel, it is required to:

1. Determine what properties such a benchmark must possess to be sensitive to that interference channel.
2. Execute and perform measurements on the 'sensitive' benchmark on one core, while other cores are idle, to establish a baseline when there is no interference.
3. Identify the properties a benchmark must possess to be aggressive on the interference channel.
4. Execute and perform measurements on the 'sensitive' benchmark on one core, while the 'aggressive' benchmarks are run on the other cores.
5. Compare the timing properties of the 'sensitive' benchmark with and without interference.

3) Mitigation mechanism identification

In parallel with the hardware resource identification, mitigation mechanisms for these interference channels should be identified. Different mitigation levels are possible:

- Hardware configuration. For example, it may be possible to mitigate an interference channel related to cache evictions by configuring cache partitioning on hardware platforms that support it. Alternatively, hardware devices and features may be disabled to remove some interference channels.
- Integration-level configuration. For example, a specific data/code placement in memory could be configured to enforce spatial partitioning for some resources. Time partitioning at scheduling level can also be used to achieve exclusion between some tasks' executions and thus avoid concurrent access to some resource.
- Software architecture. For example, timing exclusion can be ensured by creating exclusion groups, either through sLET intervals re-design or by adding additional timing constraints using fractional clocks.

D. Multicore Time Budget evaluation

Once characterized, the identified interference channels need to be accounted for to derive multicore TBs in isolation. This can be done by mitigating the interference, or upper-bounding its maximal impact to add it to the single-core TB.

1) Multicore interference impact evaluation

Results from the interference channel characterization can be used to assess whether the impact is sufficiently small for the interference channel to be neglected. For interference

channels that cannot be neglected, their actual impact on the different tasks needs to be assessed. Indeed, interference may not have an impact for all interference channels, depending on the actual use of hardware resources by the different tasks.

As for the hardware characterization step, the interference impact evaluation should be conducted on target:

1. First, each sLET task is executed in isolation on one core, with some instrumentation for timing and resource usage, while other cores remain idle.
2. Then, the list of interference channels against which the task should be characterized is refined, removing the ones related to resources the code won't exercise.
3. Finally, the task is executed again on one core, while exercising the remaining interference channels (using the same combinations of 'aggressive' benchmarks as for the hardware characterization).

Comparing for each EA the distributions of execution times measured with and without the 'aggressive' benchmarks provides the applicant with qualitative and quantitative information which allows the identification of: (i) interference channels of concern, i.e., ones that can actually cause time interference due to their use by the application, and (ii) EAs making significant use of each identified interference channel.

Results from the hardware characterization and the interference impact evaluation can be used to discriminate among identified channels, between: (i) those for which the impact is acceptable (in terms of safety but also performance [23] for safety-critical systems), and (ii) those for which mitigation is required. Indeed, full mitigation for all interference channels is impossible in practice, except for very simple applications [29]. Nevertheless, through A(M)C 20-193, the goal is not to reach total freedom from interference, but rather to demonstrate upper bounds on the possible impact of time interference, and that safety is not impacted. A quantitative criterion (e.g., statistical) or an empirical observation may be used, as proposed in [30]. Note that the exact meaning of 'significant' is to be defined by the applicant regarding their needs, as it is an integral part of the argumentation process and highly dependent on the use-case.

2) Multicore time interference mitigation

Mitigation strategies, such as spatial isolation or temporal exclusion using exclusion groups, should be enforced for those interference channels with the most 'significant' impact on processing time, or for high-criticality tasks (e.g., DO-178C/ED-12C DAL-A). For example, for all EAs impacted by the same interference channels, an exclusion group could be constructed by adding additional fractional TSPs so that those EAs can no longer be executed simultaneously. Note that dealing with EAs allows a finer granularity than working at the task level: indeed, a task may not access a given shared resource in all its EAs, and thus some of them could be executed in parallel with other tasks' EAs.

Once implemented, mitigation strategies should be validated. For spatial isolation, a new interference impact evaluation could be conducted; as for temporal exclusion, this may be only tested on the final integrated system.

3) Multicore Time Budget

Finally, the impact of the non-mitigated interference can be bounded for each EA, thanks to the interference impact evaluation step. This upper-bound can then be added to the single-core TB to account for MCP interference. As for the single-core case, a safety margin might be added to get the final multicore Time Budget for each EA.

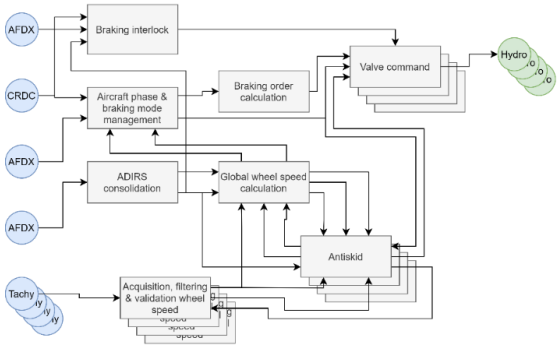

Figure 3: Functional architecture of the LGS application.

E. Multicore Time Budget validation

Eventually, measurement campaigns on the final configuration should be conducted to validate that the computed multicore Time Budgets are actually upper-bounds (all measurements for an EA in the integrated system should be less than the multicore TB derived from the previous step).

For this step, all the application's tasks must be considered at the same time. In case of pre-emptions, (i) the maximum number of times each task can be pre-empted should be evaluated, and (ii) an upper-bound on the overhead the task might experience due to a pre-emption should be computed.

VI. Application to a Case Study

To illustrate the approach, we consider an industrial use case from Safran Electronics & Defense. This application work has been conducted as part of the ARCHEOCS project. For our case study, we focus on a single interference channel.

A. Presentation of the use-case

The use-case consists of a simplified Landing Gear System (LGS), in charge of the aircraft main undercarriage. As depicted in Figure 3, it has 5 functional chains:

- One duplicated acquisition and command chain per side of the undercarriage, to get the wheel speed and apply the braking order on the hydraulic valves.
- The main chain to compute the braking command.

The different functions are all executed periodically, but at different rates: from 1Hz (for the braking order calculation function) up to 10Hz (for the acquisition part).

The LGS is deployed over an NXP T1042 multicore hardware platform consisting of four e5500 PowerPC cores running at 1.4GHz, with private L1 caches, split between instructions and data, and a unified L2 cache per core. An interconnect (CoreNet) is used to access a shared 4GB DDR4 memory, as well as several peripherals and accelerators.

B. Tools to support a full time-aware strategy

To support an application of the TB evaluation approach on the LGS, we use **ASTERIOS** as our integration solution and Rapita's tools to help with the TB evaluation process.

1) ASTERIOS solution

The **ASTERIOS** solution is developed and commercialized by ASTERIOS Technologies (formerly Krono-Safe), based on a technology from the CEA (French

Atomic and Alternative Energies Research Organization). It offers a time-aware methodology, supported by a set of industrials tools, to develop safety-critical embedded systems.

ASTERIOS is centered around an implementation of the sLET model as the PsyC language. It comes with a dedicated toolchain to (i) help with the design and configuration of a PsyC (i.e., sLET) application and (ii) support the compilation for a given hardware target. EA timing constraints (i.e., sLET intervals), extracted from the PsyC design, once mapped to physical time, and Time Budgets, provided by the user for a specific hardware target, can be used as inputs for automatically computing a feasible static schedule (if any) thanks to the **ASTERIOS** toolchain. To support and enforce sLET execution at run time, **ASTERIOS** provides a certified target-specific real-time microkernel which implements time and space partitioning. In particular, it ensures that the schedule generated by the toolchain is not violated at run time (i.e., that no EA exceeds its TB): a run time mechanism is able to detect any violation to prevent the offending task (or the whole application) from continuing its execution, as neither timing nor functional determinism can thereafter be ensured. Finally, **ASTERIOS** offers a qualified tool to verify that the toolchain's outputs are compliant with the user's input (and in particular the specified sLET design) [31].

2) Rapita's solution

Rapita Systems provides a tool suite, called Rapita Verification Suite (R**VS**), to support verification of critical aerospace and automotive systems. From a multicore timing perspective, it allows users to: (i) analyze and verify scheduling behavior on-target using Rapi**Task**, (ii) analyze and verify software timing behavior on-target down to the basic block level using Rapi**Time**, (iii) automate test harness generation using Rapi**Test**, and (iv) perform testing that exercises specific multicore interference channels using Rapi**Daemons**. Where applicable, these tools are available with DO-330/ED-215 qualification kits.

R**VS** can be used as a key part of Rapita's **MACH**[178] solution for certifying multicore aerospace projects in accordance with DO-178C/ED-12C and A(M)C 20-193. **MACH**[178] comprises several components, including software tools with associated qualification kits; procedures, templates, and checklists for generation of multicore certification evidence; an IP library covering interference channels in popular avionic multicore processors; and specialist engineering and consultancy services. The **MACH**[178] procedures both directly address A(M)C 20-193 objectives related to multicore timing, but also intend to provide the required supporting evidence. For example, if debug performance counters are used to provide evidence that: (i) a tool is performing correctly; or (ii) some software is not accessing a particular hardware resource, then these counters also need validation. Therefore, an event monitor validation procedure is incorporated.

3) Tools integration

The LGS software is integrated on the T1042 platform using **ASTERIOS**. Each function is mapped to a PsyC task. Two additional tasks are added for logging. All inter-task communications are performed through sLET communication channels. At this point, the logical and functional behavior of the PsyC application, in particular the data/control coupling, can be verified offline (i.e., without a compilation and execution on the T1042) thanks to the dedicated **ASTERIOS** simulator. Specifically, it allows verification that worst-case dataflow latencies resulting from the application's timing architecture (according to the visibility principle) are compatible with the high-level end-to-end requirements.

For the LGS application integration on the T1042, a static allocation of the tasks to the CPU cores is used. The main chain tasks, as well as the two logging tasks (one per core), are allocated to Cores 0 and 1. The duplicated chains are allocated to Cores 2 and 3 (2 chains per core). All tasks have access to the shared DDR4 memory. Moreover, all caches are enabled and a write through policy is set for the data cache.

As each task is strictly periodic, we consider one single Time Budget for all EAs of a same task. For TB evaluation, Rapita's tools have to be used with **ASTERIOS**. A connection has been prototyped as part of the ARCHEOCS project: (i) an interfacing layer allows R**VS** to derive **ASTERIOS**-relevant timing results, which means in particular computing timing estimates for each EA, and (ii) Rapi**Daemons** can be run against an **ASTERIOS** application on dedicated CPU core(s) without altering scheduling on other core(s).

C. Single-core Time Budget evaluation

As presented in Section V, TB evaluation is conducted on each task in isolation. To stub the communications from and to that task, an additional task is added to act as a 'mock' producer and receiver. It is allocated to the same CPU core as the task under analysis, to avoid creating multicore interference, and its timing behavior is designed to match exactly the one of the task under analysis (i.e., same period, as all LGS tasks are strictly periodic), to avoid any pre-emption.

Moreover, to conduct measurements, a valid schedule is required for the task in isolation. So, an initial TB has to be provided. This presents a cyclic dependency, as the goal of this initial schedule is to perform measurements that will allow an actual TB to be derived. To overcome this issue, and as each task is run in isolation, oversized TBs can be used for the sole purpose of generating a valid schedule. Another solution is to use the concept of ambivalent logical clock, which is implemented in **ASTERIOS**. An ambivalent clock can map logical time to physical time but can also switch to purely logical execution (i.e., regardless of physical time). Since ambivalent clocks are logical clocks, the execution of the scheduled tasks remains correct: logical ordering is preserved, only their physical timing constraints are altered. Thus, ambivalent clocks are definitely not suitable for production systems, but can be used to logically execute a whole system on a hardware target by relaxing the TBs constraints at run time: if a TB is exceeded, then the ambivalent clock allows for the corresponding EA to complete its execution by temporarily delaying any activation of other EAs.

The task's code is instrumented to capture timing and resource usage information on each activation and termination of an EA of the task. The maximal observed Time Budget estimates (i.e., HWMs) for a few tasks are summarized in Table I. There is quite a large variability in execution times among the different tasks, from a few µs to more than 1ms.

D. Hardware characterization

1) Hardware resource identification

The output of the hardware resource identification step should be a complete list of the hardware components in the platform. This can then be used to check that all relevant hardware resources have been adequately analyzed and

characterized. Additionally, this activity can provide an early indication of whether there is adequate documentation available for the platform to support further analysis.

For our case study, we focus on a single resource, the T1042's shared DDR memory. As it is used by all tasks for instructions and data (including stacks), this is likely to be a major interference source for the application. In a typical DDR controller, there are several interference channels. We focus only on the one concerned with competition for rowbuffers. A DDR memory device stores data in 'rows', which in the T1042 are 8KiB long. When data is requested from a particular row, the DDR controller performs a destructive read on the entire row and buffers it in a rowbuffer. While the row is in the buffer, many reads and writes may be performed on that row. When the row has been finished with, the buffer is needed for another transaction, or a timeout has been reached, it is written back into the DDR device. On many multicore platforms, this interference can cause a significant increase in execution time.

2) Interference channel characterization

To evaluate the maximal possible impact of rowbuffer interference, we use a specifically tailored Rapi**Daemon**, targeting the DDR controller rowbuffers, as our interference benchmark. The T1042's DDR controller contains 64 rowbuffers: up to 64 rows may be buffered at a time, but accesses to unbuffered rows require that a currently-buffered row is written back to the DDR memory device before the new row can be accessed. So, the Rapi**Daemon** used in this case study is designed to cause a row eviction with every instruction executed; this should be able to demonstrate the worst-case effect of contention for rowbuffer availability.

For the interference channel characterization, we create a specific task, referred to as the unit under test (UUT), to act as a benchmark for the analysis. It executes the Rapi**Daemon** code on Core 0 at a 10Hz frequency. Rapi**Daemons** on other cores (to create interference), are executed continuously in bare metal. For the measurements, we consider 4 scenarios:

1. No Rapi**Daemon** is run in parallel with the UUT.
2. 1 Rapi**Daemon** is run in parallel on Core 1.
3. 2 Rapi**Daemons** are run on Cores 1 and 2.
4. 3 Rapi**Daemons** are run on Cores 1, 2 and 3.

Timing measurements are retrieved for each execution of the UUT under each scenario. From those measurements, TB estimates accounting for rowbuffer interference are computed for each execution of the UUT, using the R**VS** tools with the dedicated **ASTERIOS** interfacing layer. As depicted in Figure 4, the impact of interference can be quite high: up to a 43% increase for the HWM when suffering from interference due to Rapi**Daemons** running on all three remaining cores.

3) Mitigation mechanisms identification

Different mechanisms provided by **ASTERIOS** can be used for mitigation. At design level, we can use fractional

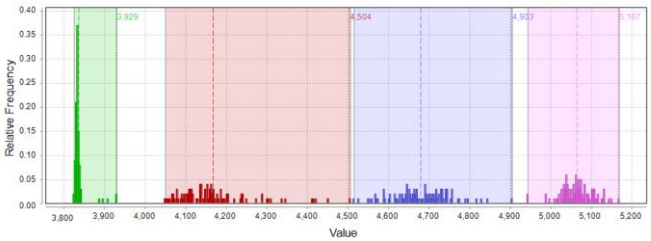

Figure 4: Histograms of execution times (in µs) computed by RVS for the hardware characterization under the different scenarios (green: isolation; red: 1 RapiDaemon; blue: 2 RapiDaemons; purple: 3 RapiDaemons).

TSPs to construct exclusion groups between some EAs. At integration level, we can rely on the **ASTERIOS** toolchain's frames exclusion mechanism, which allows the automatic computation (if possible) of a static schedule enforcing a temporal exclusion between some EAs specified by the user. Note that spatial partitioning is not considered as it would be more cumbersome to implement for a shared-memory architecture like the T1042. For other architectures using memory hierarchies (with some shared memory and other local to the CPU cores), this could be a sustainable solution.

E. Multicore Time Budget evaluation

1) Multicore interference impact evaluation

As competition for rowbuffers can be a potentially significant interference channel, its actual maximal impact on the LGS tasks needs to be quantified. This time, we deal with each task as the UUT and we consider the same 4 Rapi**Daemons** configuration scenarios as previously.

HWMs for the worst-case scenario (3 Rapi**Daemons)** are synthesized in Table I. As all tasks access the DDR, there is always some interference when running contender code. But its impact differs a lot: some tasks of the main chain suffer from overhead of several dozen µs (compared to a few µs for the other tasks). As this impact is larger than the HWMs of most tasks, we chose to consider them as part of an exclusion group for which some mitigation should be implemented.

2) Multicore interference mitigation

For the LGS case study, we consider 2 different mitigation means serving different purposes. First, we deal with the 2 logging tasks which both share a common resource (the logging mechanism). As they can be executed in parallel, this could lead to a functional interference. So, a temporal exclusion between their EAs is enforced, using fractional TSPs. Then, we consider the set of tasks that can suffer significantly from interference over the DDR4 memory, identified in the previous step. As those tasks have very different rates, implementing temporal exclusion through fractional TSPs might be quite hard and over constraining when generating the static schedule. Thus, we rely on the **ASTERIOS** toolchain's frames exclusion mechanism to generate a static schedule ensuring the temporal exclusion.

Table I: Timing results for LGS tasks.

Task	**Single-core evaluation**	**Multicore interference evaluation (3 RapiDaemons)**	**Overhead**	**Multicore final integration (with mitigation)**
ADIRS consolidation	1397.0µs	1455.2µs	+58.2µs	1492.0µs
Aircraft phase & braking mode management	1255.3µs	1306.2µs	+50.9µs	1316.3µs
Braking order calculation	1264.6µs	1310.4µs	+45.8µs	1279.6µs
Global wheel speed calculation	1279.0µs	1325.2µs	+46.2µs	1294.6µs
Other tasks	12.5µs-115.9µs	13.5µs-120.7µs	+0.9µs-5.3µs	21.6µs-260.0µs

3) Towards multicore Time Budgets

To derive safe multicore TBs (in the context of a certification project), all possible interference channels should be dealt with. This can be seen from the measurements conducted on the final configuration (i.e., integration of all the LGS tasks on the T1042). As depicted in Table I, HWMs for all tasks are larger than the ones observed for the interference evaluation step. Thus, there are clearly other interference channels that should be characterized and accounted for. Note that, for the mitigated tasks, the increase in execution times remains quite small (less than 10%). So, the impact of those other interference channels could be accounted for as an additional safety margin on the multicore TB.

VII. Summary and perspectives

In this paper, we described a time-aware strategy suitable for safety-critical real-time systems, based on the sLET paradigm. We showed that sLET properties can help a DO-178C/ED-12C applicant build an argumentative strategy for answering A(M)C 20-193 objectives related to functional and time interferences. Thanks to sLET, functional interference is fully prevented by design, and time interference can be restrained through temporal exclusion.

The application of a sLET-based strategy to an industrial use case has been illustrated using the **ASTERIOS** solution, which is already being deployed by Safran Electronics & Defense for single- and multicore commercial systems. To meet the required A(M)C 20-193 objectives, we showed that Rapita's approach and tools can support interference and timing analyses for **ASTERIOS**-based systems.

In future steps, we plan to further develop our multicore Time Budget evaluation methodology to provide a comprehensive solution for implementing and integrating safety-critical real-time systems on MCPs.

References

[1] S. Gerhold, M. Dunham and B. Sletteland, "Alternative multi-core processor considerations for aviation," 2018.

[2] C. Maiza *et al.*, "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems," *ACM Computing Surveys,* 2020.

[3] European Union Aviation Safety Agency, "AMC 20-193 Use of multi-core processors," 2022.

[4] Federal Aviation Administration, "AC 20-193 - Use of Multi-Core Processors," 2024.

[5] S. Baruah and G. Fohler, "Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems," *RTSS,* 2011.

[6] A. Benveniste *et al.*, "The synchronous languages 12 years later," *Proceedings of the IEEE,* 2003.

[7] A. Yip *et al.*, "The ForeC Synchronous Deterministic Parallel Programming Language for Multicores," *MCSOC,* 2016.

[8] J. Forget *et al.*, "A Multi-Periodic Synchronous Data-Flow Language," *HASE,* 2008.

[9] J. Forget, Un Langage Synchrone pour les Systèmes Embarqués Critiques Soumis à des Contraintes Temps Réel Multiples, 2009.

[10] C. Pagetti *et al.*, "Multi-task Implementation of Multi-periodic Synchronous Programs," *Discrete event dynamic systems,* 2011.

[11] T. A. Henzinger, B. Horowitz and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *LNCS,* 2003.

[12] E. Farcas *et al.*, "Transparent distribution of real-time components based on logical execution time," *LCTES,* 2005.

[13] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," *Advances in Real-Time Systems,* 2012.

[14] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems,* 2008.

[15] F. Boniol *et al.*, "Modelling and analyzing multi-core COTS processors," *ERTS,* 2022.

[16] C.-G. Lee *et al.*, "Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling," *IEEE Transactions On Computers,* vol. 47, no. 6, pp. 1-14, 1998.

[17] V. David *et al.*, "Safety properties ensured by the oasis model for safety critical real-time systems," *SAFECOMP,* 1998.

[18] F. Siron *et al.*, "The synchronous Logical Execution Time paradigm," *ERTS,* 2022.

[19] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM,* 1978.

[20] C. André, F. Mallet and M.-A. Peraldi-Frati, "A multiform time approach to real-time system modeling; Application to an automotive system," *SIES,* 2007.

[21] M. Lemerre and E. Ohayon, "A Model of Parallel Deterministic Real-Time Computation," *RTSS,* 2012.

[22] P. Axer *et al.*, "Building timing predictable embedded systems," *ACM Transactions on Embedded Computing Systems,* 2014.

[23] X. Jean, L. H. Mutuel and R. Soulat, "Assurance of Multicore Processors: Limits on Interference Analysis," *FAA Final report,* 2020.

[24] J. Bin *et al.*, "Studying co-running avionic real-time applications on multi-core COTS architectures," *ERTS,* 2014.

[25] J. Guyomarc'h *et al.*, "Non-Simultaneity as a Design Constraint," *TIME,* 2020.

[26] J. Cavicchio and N. Fisher, "Integrating Preemption Thresholds with Limited Preemption Scheduling," *RTCSA,* 2020.

[27] S. H. VanderLeest and D. C. Matthews, "Incremental Assurance of Multicore Integrated Modular Avionics (IMA)," *DASC,* 2021.

[28] P. Radojković *et al.*, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization,* 2012.

[29] F. J. Cazorla, J. Abella and E. Mezzetti, "Dissecting Robust Resource Partitioning, Robust Time Partitioning, and Robust Partitioning in CAST-32A," *SAE,* 2021.

[30] A. Ferlin, E. Jenn and M. Kaufmann, "Accounting for interferences in the design of Time-Triggered Applications," *ERTS,* 2020.

[31] A. Methni, E. Ohayon and F. Thurieau, "ASTERIOS Checker: A Verification Tool for Certifying Airborne Software," *ERTS,* 2020.