A Refinement Method for Interference Analysis using the PHYLOG Modeling Language

Guillaume Brau, Eric Jenn, Emmanuel Courty, Kevin Delmas, Frédéric Boniol

A Refinement Method for Interference Analysis using the PHYLOG Modeling Language

Guillaume Brau, Eric Jenn
IRT Saint-Exupéry
Toulouse, France

Emmanuel Courty
IRT Saint-Exupéry &
Liebherr Aerospace
Toulouse, France

Kevin Delmas, Frédéric Boniol
ONERA
Toulouse, France

***Abstract*—Temporal interference may occur in multicore processor systems due to tasks running in parallel competing for shared resources such as buses or memories. This paper presents a model-based interference analysis based on the PHYLOG framework that intends to help in the certification process of multicore aeronautical systems. As PHYLOG does not define a clear modeling method, a refinement approach is proposed to model the system using the PHYLOG Modeling Language (PML). Our objective is to define a process that enables to build a model that is both precise and reliable so that analysis results are sound. The approach is finally validated on an industrial use case from the aerospace domain.**

I. Introduction

In the last decade, multicore processors have become the norm in the general market. However, their use in critical real-time systems still represents a challenge since those systems have to meet strong temporal requirements with a high level of confidence and have to comply with regulatory requirements.

In order to demonstrate compliance with temporal requirements, upper bounds on execution times accounting for all contributions at hardware and software levels must be estimated. In the avionics domain, for instance, the AMC 20-193 [1] standard requires that temporal interferences which may occur when tasks running in parallel access shared resources (e.g., caches, memories, buses) are addressed. For example, the *MCP_Ressource_Usage_3* (RU3) demands that: "the applicant has identified the interference channels that could permit interference to affect the software applications hosted on the multicore processor cores, and has verified the applicant's chosen means of mitigation of the interference".

Towards that goal, model-based approaches can be used to help in the certification process of aeronautical systems that use multicore processors. These solutions build on a language that allows to describe the system and an analyzer to infer properties about it: one can e.g. combine the AADL language to describe the architecture of the system and Prolog to identify interferences [2], or use the LNT formal specification language to capture the system's behavior and the CADP toolchain to detect interferences [3]. However, methodological aspects to use these tools remain mostly unaddressed and must be further defined in order to be used in an industrial context: *How to build a relevant model of the system? What to learn from it? How to use it in a certification process?*

In this paper, we focus on interference analysis using the PHYLOG approach [4]. In PHYLOG, the system architecture is to be described using the PHYLOG Modeling Language (PML), which model is then analyzed in order to identify interferences. Hence, the construction of the PML model is a critical point when applying the approach: *How to build the PML model?* In this paper, we propose to apply a refinement process in order to build an accurate and reliable model of the system. This process implements different types of refinements: structural (i.e., improving the description of the components of the architecture such as memories and buses), temporal (e.g., integrating data from the scheduling plan), etc. Our method is applied to an industrial use case from Liebherr, which is deployed on the AURIX TC399XE platform.

This paper is organized as follows: Section II deals with related work. We briefly introduce the use case in Section III. The refinement method is presented in Section IV with PML views implementing the method in Section V and application to the use case in Section VI. Finally, we discuss our approach (Section VII) and conclude with possible perspectives (Section VIII).

II. Related Work

The interference problem is identified e.g. in [5] as "alterations of the processor's behavior seen by software running on one core due to activities ordered by software running on other cores.". Thus, certification authorities such as EASA and FAA notably set interference-related objectives for the certification of multicore systems [6], [1]. Focusing on the argumentation to reach these objectives, [4] propose to organize an argumentation strategy into a series of *evidences* (or sub-claim) towards the *claim* (the objective to demonstrate). They show in particular that the aforementioned RU3 objective involves identifying all interferences (*sub-objective 1*) and to classify their effects (*sub-objective 2*). We review some related work on these topics.

a) Evaluation of interference effects: A first class of works aims to evaluate the impact of contentions occurring within shared hardware resources on the applications execution times. For instance, [7] considers a multicore processor architecture composed of a single bus providing access to a shared memory, and it proposes a method to determine an upper bound on the number of bus requests that software tasks can generate

in a given time interval. Both [8] and [9] focus on measurement techniques based on dedicated stressing benchmarks and hardware monitors to characterize the architecture and the shared resources that can cause interferences between software applications.

b) Identification of interferences: Another class of works relies on a formal model of the architecture and a formal analysis method to explore the set of interference channels. A previous work by Brindejonc et al. [10] proposes a way to characterize the interference behavior and to identify interference channels on a multicore processor. Their approach consists in identifying a set of test classes that completely covers the interferences that could occur in the architecture. However, they consider multicore architectures as black boxes, thus ignoring internal components of the architecture. In [2], the authors propose a tool ("Strange") that identifies interferences using an AADL structural model of the SoC that is first translated into a set of (Prolog) facts that is queried using a Prolog program. Several works have tried to circumvent the limitations of a pure structural representation of the SoC. In [3], for instance, the authors use LNT to capture the behavior of the Infineon TC275's crossbar arbiter and exploited the CADP toolchain to detect interferences. Two approaches have been investigated: PATCHECK, based on the detection of predefined patterns showing the manifestation of interference (i.e., the interleaving of a request concerning core Y in the sequence of transaction concerning core X), and SYNCHECK, based on the comparison of traces obtained in isolation and in contention. These two methods have been applied on a small part of the SoC, and their scalability has not been demonstrated.

Finally, note that the previous works either address sub-objective 1 (identification of interferences) or sub-objective 2 (evaluation of interference effects). In other words, both objectives are addressed *separately*, without stating whether they are related and how. In our approach both activities act in a complementary way: the PML model is used to identify the set of interference scenarios that must be evaluated; evaluation, in turn, enables to estimate the impact of the scenarios on the software and if this impact is compliant with the constraints defined by the applicant. Thus, the construction of the PML model is driven by the evaluation: the PML model must be refined until interference effects fulfill the constraints (see Section IV).

III. Use Case

Hereafter, we successively present the software and hardware parts and the execution environment of the industrial use case developed by Liebherr Aerospace (referred to as LTS use case in the following).

a) Application: The software parts consist of two distinct legacy applications as well as a common board support package (BSP) software:

- the Integrated Air Management System (IAMS) that manages air bleed, air conditioning and cabin pressure control,
- the Power Electronics (PE) system that controls electrical motors,
- the common BSP software that deals with Inputs/Outputs (I/Os), e.g., GPIO, PWM, etc.

The IAMS and PE sub-systems do not coexist on the same hardware platform in the actual system. However, we deploy them on the same hardware target in order to (i) address different typologies of timing constraints (20 Hz for the IAMS, 20 kHz for the PE), (ii) increase the pressure on the use of shared resources provided by the platform, and (iii) investigate the integration of multiple heterogeneous applications on the same computation platform.

b) Hardware platform: The application software is integrated in a 6-cores Infineon AURIX TC399XE processor [11], which simplified architecture is depicted in Figure 1. The AURIX has been designed in order to minimize possible hardware interferences: each core has dedicated program (PSPR, PFLASH) and data (DSPR, DLMU) memories, 3 shared memories (LMUs) can be used by the cores through a crossbar (SRI), and a shared bus (FPI) is provided to access I/Os.

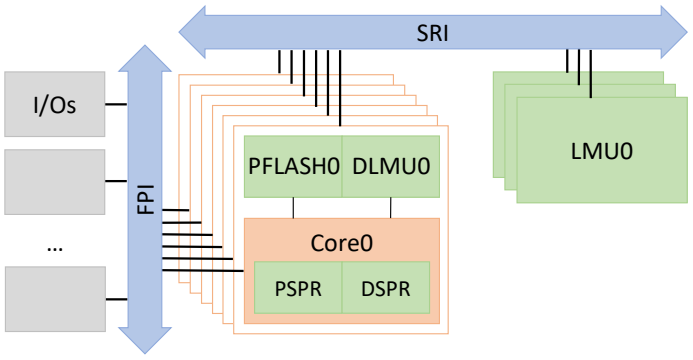


Fig. 1: AURIX TC399XE platform (simplified view).

c) ASTERIOS: The ASTERIOS toolchain and execution environment, provided by ASTERIOS Technologies [12], is used to integrate both applications and the common BSP on the AURIX platform and enforce the timing constraints.

Applications are developed using PsyC [13], a software architecture description language based on the synchronous Logical Execution Time (sLET) paradigm [14]. In this model, each new activation of a task is constrained by an earliest start date and a deadline specified by the user based on logical clock ticks. The IAMS application is implemented with two PsyC tasks (ag_iams and ag_cpcs), the PE application includes two tasks (ag_fast and ag_slow) and the BSP three tasks (worker_gpio, worker_adc and worker_pwm).

The PsyC design is then compiled, together with the user application code, into an executable binary that is executed by the ASTERIOS's real-time micro-kernel. At run time, the kernel relies on a static schedule generated by the ASTERIOS toolchain, called Repetitive Sequences of Frames (RSF), to enforce the tasks timing constraints and determinism (in multicore architectures, one RSF is generated per core). An RSF is divided into intervals, and each task may be given at most a frame within each interval for its execution (Figure 2). A frame corresponds to the CPU time allocated to a task, and is computed from the time budget provided by the user.

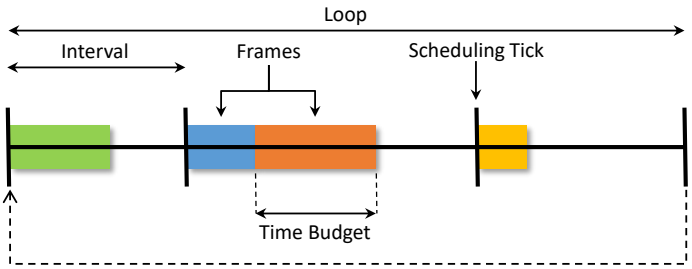


Fig. 2: Repetitive Sequences of Frames (RSF) in ASTERIOS.

IV. Approach

This section presents the refinement approach that is based on the PML language.

A. PHYLOG Modeling Language

The PHYLOG Modeling Language (PML) provides a set of constructs capturing the concepts identified in the AMC 20-193 and supporting interference analysis. Here, we briefly introduce the main elements of the language, further details can be found e.g. in [4].

a) PML model: A system can be described in PML in two main *views*: structural and behavioral.

The *structural* view describes the system architecture with:

- platform, i.e., the set of hardware components (cores, memories, communication resources, etc.) and physical connections between them,
- software, i.e., the software applications and their data,
- the allocation of the software on the platform : applications are allocated on cores and data on memories,
- routing aspects, e.g., paths from sources to targets.

The *behavioral* view details transactions (core accesses to shared resources, e.g. memory read/write) occurring on the platform and provides additional interference specifications.

b) Semantics of a PML model: Following [15], the platform in a PML model describes *initiators* (e.g., a core), *targets* (e.g., a memory component), and *transporter* (e.g., a bus). Applications are allocated to initiators and data to targets. Hardware components provide one or several *services* (e.g., store or load). When an application requests to access or write a data, it will trigger a *single transaction*, i.e., a sequence of components' services used to fulfill the request. For instance, to load a data from the *LMU0*, an application running on the *Core0* will trigger the single transaction: $Core_0^{ld} \cdot SRI^{ld} \cdot LMU_0^{ld}$ where C^{ld} stands for *load service of component C*. Several single transactions can be triggered by the applications executed by the initiators. A set of concurrent single transactions is called a *multi-transaction*. For instance, the multi-transaction $\tau = (Core_0^{ld} \cdot SRI^{ld} \cdot LMU_0^{ld} \| Core_1^{ld} \cdot SRI^{ld} \cdot LMU_0^{ld})$ is the concurrent access by *Core0* and *Core1* to the *LMU0* through the *SRI*. The set of all possible multi-transactions, denoted $\mathcal{T}$, of a given PML model M is then all the possible concurrent single transactions that can be triggered by the applications executed by the initiators. An interference occurs when the simultaneous usage of a set of services may impact timing behavior. A typical case is the simultaneous use of a single service by several single transactions in the multi-transaction. For instance, in the multi-transaction τ, the load services of the *SRI* and *LMU0* are used concurrently by the two cores. The purpose of the PML analyzer is then to identify efficiently the multi-transactions that may produce an interference.

B. Refinement Approach

In our approach, following AMC 20-193 objectives, a PML model is used as an appropriate abstraction of the system to (1) identify a set of interferences, and then (2) evaluate the effects of interferences.

a) Interference identification: Let M a PML model, interference identification is the analysis applied over the PML model $Id(M)$ that classifies multi-transactions $\mathcal{T}$ described in M into $\mathcal{I}$, interfering multi-transactions, and $\mathcal{F}$, non-interfering multi-transactions.

As explained in Section IV-A, an interference in PML comes from the concurrent use of a service (or exclusive services) provided by a hardware component within a multi-transaction. An interference involves n ($\geq$ 2) transactions, so we talk about n-ary interference (itf-n for short). The PML analyzer identifies the multi-transactions which may compete for services provided by the components of the platform (*interference scenarios*) and the components where those conflicts occur (*interference channels*). Complementary, non-interfering scenarios (non-interfering multi-transactions) are also provided. The PML analyzer relies on MonoSat solver [16] to compute interferences.

b) Evaluation: Once interference scenarios have been identified, their effects on the software is evaluated in order to check compliance with the constraints defined by the applicant. For instance, the constraints may relate to real-time, requiring to evaluate response times in order to check that the tasks meet their deadlines (see Section VI-B for an example of evaluation).

c) Refinement process: The PML model being the main source of information to perform interference analysis, it shall satisfy the following properties:

- *Coverage*: the model enables to identify all interferences,
- *Precision*: the model enables to identify only interferences.

Coverage impacts safety, so it is explicitly required by the AMC 20-193. *Precision* impacts cost, as it may lead to unnecessary verification activities and over-design.

Assuming an initial model that would *satisfy coverage* and a refinement process that would *preserve coverage*, the approach consists in refining the model in order to increase precision *in order to meet the constraints*. In practice, and due to the difficulty to ensure coverage *by design*, additional verification and validation activities are also required. One way to ensure coverage is for example (i) to make reasonably conservative modeling assumptions and (ii) to verify these assumptions experimentally (see Section VI-C for an example of validation).

Let us define more precisely what is meant here by refinement and what are the objectives of the refinement process.

False-positives and false-negatives: A PML model is an abstraction – thus imperfect representation – of the system that may lead to identify false-positives and false-negatives. Let I (resp. F) the set of *actual* interfering (resp. non-interfering) multi-transactions that would be observed on the target, false-positives $\mathcal{FP} = \mathcal{I} \cap F$ are multi-transactions that are erroneously classified as interfering and, conversely, false-negatives $\mathcal{FN} = \mathcal{F} \cap I$ are multi-transactions that are erroneously classified as non-interfering. The presence of false-positives means that the model is imprecise, whereas the presence of false-negatives means that the coverage objective is not fulfilled.

Refinement (and validation): Refinement is a function $R(M_i) = M_{i+1}$ that adds details on the model, thus restricting the set of the possible behaviors and increasing the accuracy of analysis. Applied to the PML model, the consequence of refinement is that reported false-positives are reduced: $\mathcal{I}_{i+1} \subset \mathcal{I}_i$ and, consequently, $\mathcal{FP}_{i+1} \subset \mathcal{FP}_i$.

The objective is therefore to apply refinement so that the number of false-positives $\mathcal{FP}$ is minimized (precision objective) in order to meet the constraints. In addition, validation of the model must ensure that all interferences are identified (coverage objective), i.e., $\mathcal{FN} = \emptyset$.

The refinement process is illustrated in Figure 3. Refinement of the PML model is represented on the horizontal axis: M_i is the model at phase i of the process (represented with a white circle in Figure 3). Interference identification is shown on the vertical axis: Id inputs the PML model M_i and outputs the set of interfering multi-transactions $\mathcal{I}_i$ and non-interfering multi-transactions $\mathcal{F}_i$. Evaluation is illustrated in the gray rectangle in the bottom left-hand corner of Figure 3: the evaluation function Ev inputs identified interfering multi-transactions $\mathcal{I}_i$ and enables to conclude whether the constraint is met (acceptable state) or not (failure state). If effects are not acceptable, the PML model can be refined in order to reduce $\mathcal{I}$ and make the evaluation more precise: in Figure 3, if $Ev(\mathcal{I}_i)$ is not acceptable, M_i is refined into M_{i+n} and evaluation is applied on the refined $\mathcal{I}_{i+n}$ set ($\mathcal{I}_{i+n} \subset \mathcal{I}_i$). The refinement process finishes when either (1) evaluated interference effects are acceptable or (2) the PML model cannot be refined anymore (in which case the system must be redesigned in order to mitigate interferences).

We explain in the following sections how the PML views can be refined (Section V) and show an application of our approach to the case study (Section VI).

V. Refinement and PML Views

This section explains how refinement can be applied to the different PML views. Taking the LTS use case for instance, we describe the initial, coarse-grained, model and illustrate different refinements applied to the structural and behavioral views.

A. Baseline Model

A PML model is a Scala application in which each file describes an aspect of the architecture through a dedicated class/interface. A PML model is thus modular, making it possible to easily analyze variants of the system, e.g., models at different refinement stages.

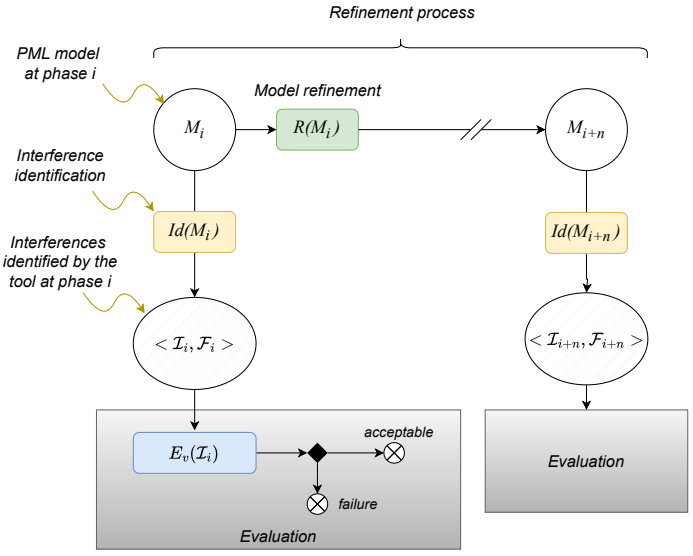


Fig. 3: Refinement process.

Modeling starts with the baseline model (M_0) that applies the most conservative modeling assumptions with information that can be easily found in the datasheet (e.g., block diagram of the platform). The initial model is as simple as possible with very little information on the parallelization capabilities of the system: no (or few) assumptions are made on the internal structure of the components, e.g., we make the assumption that the applications all execute at the same time, that transactions all take the same path, etc.

The initial model thus only describes basic structural views (platform, software and allocation) and transactions.

```
class core_unit(name: Symbol) extends Composite(name) {

  // core
  val core: Smart = Smart()

  // internal memories
  val pspr: Target = Target()
  val dspr: Target = Target()

  // connections (the core accesses its private memories)
  core link pspr
  core link dspr

}
```

Listing 1: Composite structure of a core unit.

a) Platform: The TC399XE platform used in the LTS use case embeds several core units (see Figure 1), which composite structure is described in Listing 1: a core unit includes a core (defined as an initiator with the "smart" keyword), internal memories (defined as targets), and connections between them. The full platform is described via the "AURIXPlatform" class (Listing 2) that enumerates the hardware components (core units, memories, buses, I/Os) and sets the physical links between them. The excerpt provided in Listing 2 shows the declaration of core_unit1 connected to DLMU1 (direct connection) and LMU0 (via the SRI). In the following, core$X$ will denote the core component in core_unit$X$ (core_unit$X$.core), the same

```
class AURIXPlatform(name: Symbol) extends Platform(name) {

  // core units
  val core_unit1 = new core_unit()
  // memories
  val dlmu1: Target = Target()
  val lmu0: Target = Target()
  // buses
  val sri : SimpleTransporter = SimpleTransporter()

  // connections of the cores
  core_unit1.core link dlmu1      // direct connection to the dlmu
  core_unit1.core link sri        // connection to the sri

  // connection of the lmu to the sri
  sri link lmu0
   [...]
}
```

Listing 2: Platform model of the AURIX TC399XE (excerpt).

```
trait AURIXCoarseSoftwareAllocation extends Configuration {
  self : AURIXCoarsePlatform =>

  // Tasks
  val ag_fast : Application = Application()

  // Data
  val code_ag_fast : Data = Data()              // code
  val data_ag_fast : Data = Data()              // data

  // Tasks allocation
  ag_fast use core_unit2.core

  // Data allocation
  code_ag_fast in core_unit2.pspr
  data_ag_fast in lmu0

   [...]

}
```

Listing 3: Software model of the LTS use case (excerpt).

notation will apply to designate core_unit$X$.pspr (PSPRX) and core_unit$X$.dspr (DSPRX).

b) Software and allocation: The software comprises applications (running on cores) and their data (stored in memories). The "AURIXSoftwareAllocation" class (Listing 3) first declares the tasks and the various ASTERIOS components (real-time kernel, etc.) together with their data (tasks code and data, communication data, etc.). The allocation of both applications (on cores) and their data (on memories and I/Os) is then provided. For instance, Listing 3 shows the declaration of one application ag_fast, executed on core2, and its data, stored in PSPR2 (for its code) and LMU0 (for its data).

c) Transactions and configuration: With the platform components and pieces of software defined, it is then possible to provide more information on the usage of resources. The *transaction library* lists transactions that can be performed by the tasks. A transaction declaration involves a task, a data and a type (e.g., read or write). For instance, Listing 4 shows different examples of transactions initiated by ag_fast: code loading and data read/write. One or more *configurations* to analyze can then be defined, a configuration being a particular set of transactions

```
trait AURIXTransactionLibrary extends TransactionLibrary {
  self : AURIXCoarsePlatform with AURIXCoarseSoftwareAllocation =>

  // tasks load code
  val ld_code_ag_fast : Transaction = Transaction(ag_fast read
      code_ag_fast)

  // tasks read/write data
  val wr_data_ag_fast : Transaction = Transaction(ag_fast write
      data_ag_fast)
  val rd_data_ag_fast : Transaction = Transaction(ag_fast read
      data_ag_fast)
   [...]
}
```

Listing 4: Transaction library (excerpt).

within the transaction library.

B. Refinement

Model refinement will consist in adding precision on both structural and behavioral views. Each refinement stage aims to reduce the set of interferences by removing false-positives.

We identify several types of refinements:

- *Structural refinement* is to improve the description of the components of the architecture (e.g., memories, buses),
- *Routing refinement* is to specify the paths from initiators to targets when several paths are possible,
- *Temporal refinement* is to provide temporal exclusions,
- *Quantitative refinement* is to describe resource usage.

a) Structural refinement: Structural refinement seeks to improve the description of the components of the architecture (e.g., memories, buses).

As seen in the baseline model, the SRI is modeled as a "black box", i.e., it is specified as a simple "transporter" (Listing 2). Observing that the SRI is a main contributor to interferences, we may "open" the black box and model the internal structure of the component. For example, in Listing 5, the SRI crossbar network is modeled as a composite: the SRI is made up of a set of input and output ports (modeled as transporters) where all inputs are connected to all outputs. Therefore, every input port is intended to serve a particular core and every output port is connected to a distinct memory.

b) Routing: Specifying routes may be useful, e.g., when multiple paths to a target are possible. For example, in the AURIX platform each core ($core_n$) uses a local PFLASH memory ($PFLASH_n$) that can be accessed through a direct link. Meanwhile, all PFLASHs can be accessed by all cores via the SRI. Thus, there exists 2 paths between $core_n$ and $PFLASH_n$: (1) one that uses the direct link, and (2) one through the SRI. As in fact transactions from $core_n$ to $PFLASH_n$ use the direct link, it is necessary to specify a route in order not to count them in the SRI. For this, the configuration is extended with routing constraints, e.g., to specify that the route for transactions issued by core0 targeting PFLASH0 uses the direct link between core0 and PFLASH0.

c) Temporal refinement: Timing aspects such as tasks parameters (periods, time budgets, etc.) or scheduling affect interferences. Therefore, it would be interesting to integrate

```
class xbar(name: Symbol) extends Composite(name) {

  // input ports
  val i1: SimpleTransporter = SimpleTransporter ()
  val i2: SimpleTransporter = SimpleTransporter ()
  [...]

  // output ports
  val o1: SimpleTransporter = SimpleTransporter ()
  val o2: SimpleTransporter = SimpleTransporter ()
  [...]

  // connections (all inputs connected to all outputs)
  for {
    input <- Set(i1, i2, i3, i4)
  }{
    for {
      output <- Set(o1, o2, o3, o4, o5, o6, o7, o8, o9)
    }{
      input link output
    }
  }

  [...]
}
```

Listing 5: Structural refinement: composite structure of the SRI (excerpt).

temporal information, specified in the PsyC design or in the generated RSF, in the PML representation. While PML does not directly support such description of timing aspects, we can capture temporal information with two means: slices or exclusivity clauses.

In the first method, a temporal "slice" represents a remarkable execution unit whose transactions are encoded as a configuration. Each configuration is intended to be analyzed separately before the results from the different configurations are aggregated (duplicated interferences are counted once). The second way, which is the chosen solution in the current model, is to express (temporal) exclusions in the model. For example, by analyzing the RSF of the LTS use case, we can specify in PML that worker_adc, worker_gpio, worker_pwm (core1) and ag_fast (core2) are never executed at the same time.

d) Quantification: In some cases, a transaction can have a negligible impact on a service. In that case, one can assume that the concurrent usage of this service by other transactions will not result in a significant impact (e.g., in terms of execution times) on the application. This assumption can be specified in the PML model by identifying the transactions that do not affect specific services.

VI. Application on the Use Case

This section shows an application of our approach (Section IV) and proposed refinements (Section V) to the LTS Use Case (Section III).

We illustrate the main activities – identification of interferences, evaluation and validation – with, for example, the schedulability constraint that tasks execution times (ET) must comply with time budgets (B) defined in the ASTERIOS application architecture (see Section III, Figure 2), i.e.,

$$\forall t, ET(t) \leq B(t) \quad (1)$$

A. Identification of Interferences

We build the baseline model and then apply structural, routing and temporal refinements as explained in Section V. The next paragraphs discuss the mains results (interference scenarios and interference channels) for each refinement stage and, finally, summarize results for the overall process.

a) Baseline model: The PML analyzer enables to identify n-ary interference scenarios (i.e., interference scenarios involving n transactions). Interference channels are also identified. For instance, `<rd_data_ag_fast || wr_data_worker_adc>` denotes a 2-ary interference (or itf-2) that involves rd_data_ag_fast and wr_data_worker_adc transactions. Such interference occurs when tasks ag_fast and worker_adc attempt to load (resp. store) data at the same time in LMU0 accessed via the SRI, denoted by the interference channel { lmu0_load, lmu0_store, sri_load, sri_store }. As the platform includes 6 cores, it is possible to compute up to 6-ary interferences: e.g., we count 8024 itf-2, 205142 itf-3, 3053207 itf-4 (see *Baseline* in Figure 4). LMU0 and PFLASH0, which are two of the main shared memories, and the SRI bus to access them are the main hardware components causing interferences, e.g., SRI is implicated in 6904 itf-2, LMU0 in 2589 itf-2 and PFLASH0 in 479 itf-2. I/Os and the FPI bus to access them are other components causing interferences (e.g., 1120 itf-2 for FPI, 480 itf-2 for I/Os).

b) Structural refinement: With the SRI description (*Structural* refinement in Figure 4), we observe that the number of interferences significantly decreases: -46% for itf-2, -66% for itf-3, -75% for itf-4, etc. Most of the interferences found in the SRI are eliminated, as it turns out that most of the memories serve a single core. Thus, only the SRI's ports that are used to access shared memories still experience interferences: o_9 connected to LMU0 (2589 itf-2) and o_2 connected to PFLASH0 (479 itf-2).

c) Routing: Routing has been specified for both the baseline model and the structural refinement model, i.e., without/with refinement of the SRI. In the first case (*BaselineRouting* in Figure 4), where SRI is described as a bus, we see that interference is reduced significantly: e.g., -19% for itf-2, -26% for itf-3 and -32% for itf-4. Many interferences are avoided due to transactions that are not counted in the SRI, e.g., we count -33% of itf-4 in the SRI. In the second case (*StructuralRouting* in Figure 4), we do not observe a reduction in the number of interferences itself, as the number of interferences is already reduced due to the description of the internal structure of the SRI. However, we can see the effect of routing on interference channels: conflicts now concentrate on PFLASH0 whereas interferences are reduced in the SRI, e.g., -26% of itf-4 in o_2 connected to PFLASH0.

d) Temporal refinement: As compared with structural refinement, we can see in Figure 4 (*StructuralRoutingTiming*) that temporal refinement discards e.g. 379 itf-2 (-9%), 17142 itf-3 (-25%) and 321775 itf-4 (-42%). For these scenarios,

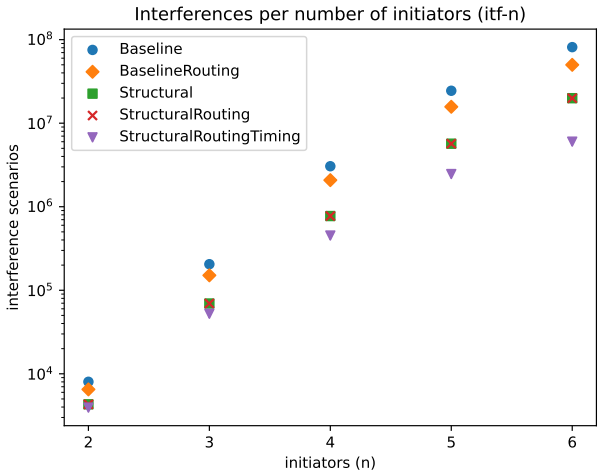
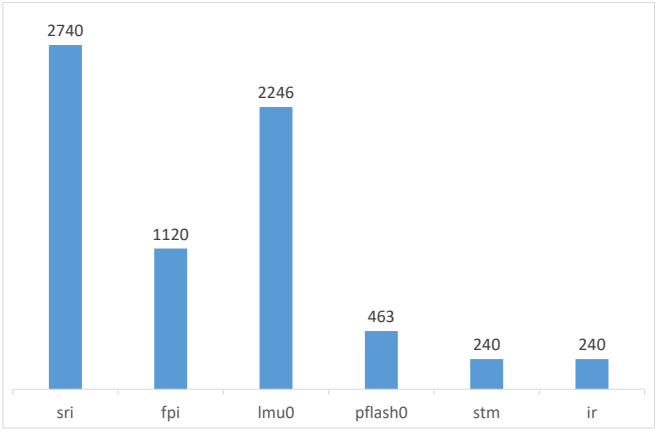
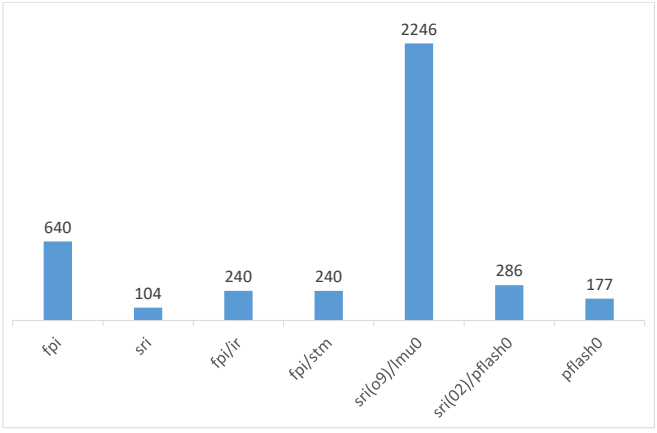


Fig. 4: Number of interference scenarios according to the number of initiators (itf-n) for different types of refinement (logarithmic scale).



(a) Itf-2 scenarios per hardware resource.



(b) Itf-2 scenarios per interference channel.

Fig. 5: Number of interference scenarios (itf-2) by (a) hardware resource, (b) interference channel.

exclusion of ag_fast and "workers" (worker_gpio, worker_adc and worker_pwm) transactions represents 52% of the total decrease, and exclusion of transactions between ag_iams and ag_cpcs 33%.

e) Summary: In total, 8976029 interference scenarios are found from the refined model: 3933 itf-2, 52272 itf-3, 451728 itf-4, 2460544 itf-5 and 6007552 itf-6. As compared to the baseline model, which displayed 109075037 interference scenarios, refinement enabled us to eliminate 92% (100099008) of interference scenarios : 51% of itf-2, 75% of itf-3, 85% of itf-4, 90% of itf-5, and 93% of itf-6. In terms of refinement steps, structural refinement, through the description of the internal structure of the SRI, has the greatest impact (83% of 100099008) before temporal refinement (17%), which specifies temporal exclusions.

We find different sources of interference (see e.g. Figure 5a for itf-2 scenarios) : memories with LMU0 and PFLASH0, inputs/outputs with STM and IR, and the buses to access them (SRI for memories and FPI for inputs/outputs). In terms of interference channels (Figure 5b for itf-2 scenarios), the SRI/LMU0 channel that is used by the cores to access the LMU0 shared memory is the main interference channel (e.g., 2246 itf-2 scenarios over 3933 itf-2 overall, that is to say 57% interference scenarios).

B. Execution Time Evaluation

After identifying the interference scenarios, we estimate execution times in order to check compliance with time budgets specified in the ASTERIOS application architecture (Equation (1)). Estimation of the execution time of a task is built on an analytical formulation of execution time that we compute with measures obtained on the target.

1) Analytical formulation of execution time: The evaluation of the execution time of a task must take into account all the interferences that the task may experience.

The execution time ET of a task t is given by:

$$ET(t) = ET_{iso}(t) + D_{itf}(t) \tag{2}$$

with ET_{iso} is the execution time in isolation (i.e., the execution time without contentions) and D_{itf} is the interference delay (i.e., the extra-time due to interferences). More precisely, the interference delay of task t is the time spent to access resources belonging to interference channels, which depends on the number of interfering accesses to every resource R in interference channels ($NA_{itf}(t,R)$) and the interfering time for each access ($DA_{itf}(t,R)$):

$$D_{itf}(t) = \sum_{\substack{R \in itf\_channel(t) \\ NA_{itf}(t,R)}} DA_{itf}(t,R) \tag{3}$$

The worst-case execution time can therefore be expressed as the combination of the worst-case execution time in isolation

$WCET_{iso}$ and the worst-case interference delay WCD_{itf}:

$$WCET(t) = WCET_{iso}(t) + \underbrace{\sum_{R \in itf\_channel(t)} WCD_{itf}(t, R)}_{WCD_{itf}(t)} \quad (4)$$

with the worst-case interference delay due to a shared resource R encompasses the worst-case number of interfering accesses to the resource $WCNA_{itf}$ and each access suffers a worst-case interference delay $WCDA_{itf}$:

$$WCD_{itf}(t,\ R) = WCNA_{itf}(t,\ R) \times WCDA_{itf}(t,\ R) \quad (5)$$

2) Application: For example, we apply Equation (4) in order to calculate worst-case execution times of `ag_fast` and `ag_slow` (parameters and results are summarized in Table I).

To do so, we first measure the execution time in isolation for each task using RVS[1] tool. For instance, Figure 6 describes the relative frequency of observed `ag_fast` execution times: observed BCET is 124 μs, and observed WCET is 134.9 μs. Therefore, we have $WCET_{iso}(\texttt{ag_fast}) = 134.9\ \mu$s

We then calculate the interference delay according to Equation (5). Access numbers (NA) are extracted from execution traces collected using TRACE32[2] tool. Figure 7 describes, for example, the frequency of accesses to LMU0 of `ag_fast` and `ag_slow`: the worst-case number of accesses (WCNA) is $WCNA(\texttt{ag\_fast}, LMU0) = 1559$ and $WCNA(\texttt{ag\_slow}, LMU0) = 4559$. In addition, hardware characterization allowed us to over-approximate access times, e.g., the maximum measured access time in contention was $WCDA_{itf}(\texttt{ag\_fast}, LMU0)$ = 23.48 ns (versus 7.53 ns in isolation).

The worst-case execution time is finally calculated according to Equation (4). The evaluation can be refined gradually, and the process stops when the calculated WCET meets the budgets (Equation (1)).

For example, as a first approximation (*level 1*), we may assume that each access to LMU0 results in an interference, i.e., for a task t, $WCNA_{itf}(t, LMU0)$=$WCNA(t, LMU0)$ with $WCNA_{itf}$ is the worst-case number of interfering accesses and $WCNA$ is the worst-case number of accesses. Otherwise (*level 2*), we may calculate the worst-case number of interfering accesses to LMU0, i.e., for two tasks t_i, t_j accessing LMU0, $WCNA_{itf}(t_i, LMU0) = WCNA_{itf}(t_j, LMU0) = \min WCNA(t_i, LMU0), WCNA(t_j, LMU0)$.

Evaluated WCETs are given in Table I together with the remaining margin to the budget. The stopping criterion is reached when the margin is positive (shown through green filled cells in the table) and unsuccessful when negative (orange filled cells). Therefore, we see that the stopping criteria is achieved by the first level of analysis for `ag_fast` and by the second level of analysis for `ag_slow`.

[1]from Rapita Systems: https://www.rapitasystems.com/products/rvs
[2]from Lauterbach: https://www.lauterbach.com/

	ag_fast	**ag_slow**
Period (us)	500	2000
Budget (us)	250.56	300
$BCET_{iso}$ (us)	124	221.5
$WCET_{iso}$ (us)	134.9	224.7
level 1:		
$WCNA_{itf}$	1559	4559
WCD_{itf} (us)	36.6	107.0
$WCET$ (us)	171.5	331.7
Margin with budget (us)	79.1	-31.7
Margin with budget (%)	31.6%	-10.6%
level 2:		
$WCNA_{itf}$	1559	1559
WCD_{itf} (us)	36.6	36.6
$WCET$ (us)	171.5	261.3
Margin with budget (us)	79.1	38.7
Margin with budget (%)	31.6%	12.9%

TABLE I: Tasks parameters and estimation of execution times.

C. Validation

Validation of the model can be reached through tests performed on the real system. As the model is aimed at identifying interferences, validation can be achieved by checking that (i) when an interference is identified from the PML model, an interference can be observed on the real system, and (ii) when an absence of interference is identified no interference can be observed on the real system. Therefore, each interfering (resp. non-interfering) scenario is associated with a test scenario.

Remind that an interfering scenario involves transactions, initiated by tasks executing on different cores, contending for one or several (exclusive) service(s) provided by a same hardware component. Conversely, a non-interfering scenario is a scenario where transactions do not contend for services provided by hardware components.

In the following, we discuss the validation of scenarios with 2 transactions, this approach can be generalized to deal with scenarios with n transactions.

a) Interference validation: Let $< tr_A || tr_B >$ an interference scenario, tr_A is initiated by $task_A$ hosted by $core_X$ and tr_B is initiated by $task_B$ hosted by $core_Y$ ($X \neq Y$), Res is the resource used to serve both tr_A and tr_B. To show that the interference scenario is valid, we can show that $task_A$ is sensitive to the service of tr_B by Res and that $task_B$ is sensitive to the service of tr_A by Res.

Consider for example interference scenarios in LMU0 involving `ag_fast`, executed on core 2, and `ag_slow`, executed on core 3. These interference scenarios can be validated by running `ag_fast` (resp. `ag_slow`) against co-runners on core 3 (resp. core 2) issuing transactions to LMU0. The objective of the co-runner is to act as the contending task, thus stressing the shared resource to maximize the bandwidth use and produce interferences. The expected outcome, if the interference scenario is valid, is an increase in the task execution time compared to the task behavior in isolation, as we can see for `ag_fast` in Figure 8 (ag_fast vs 1 contender lmu0).

b) Non-interference validation: Let $< tr_A || tr_B >$ an interference scenario, tr_A is initiated by $task_A$ hosted by $core_X$

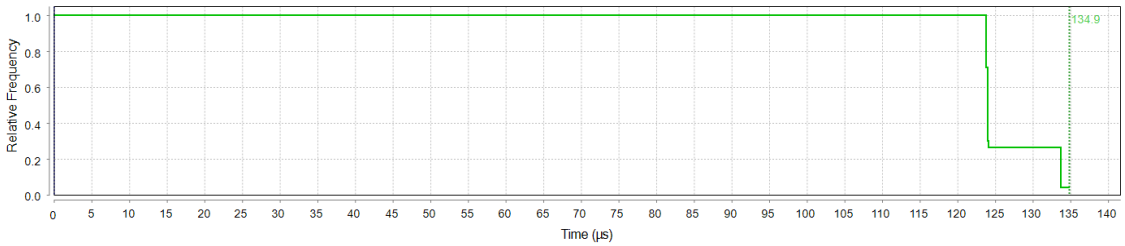
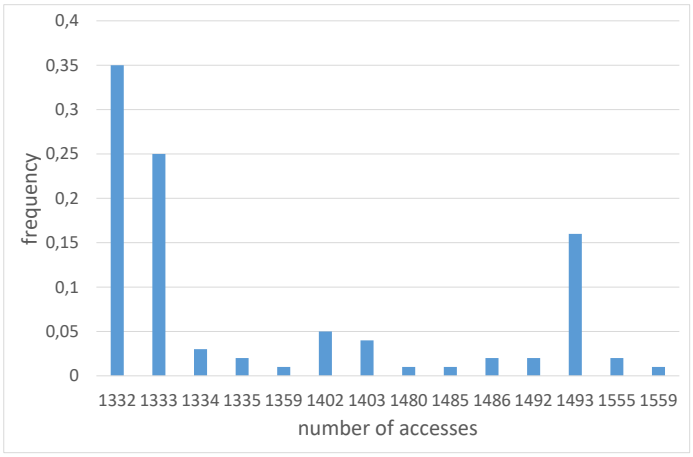
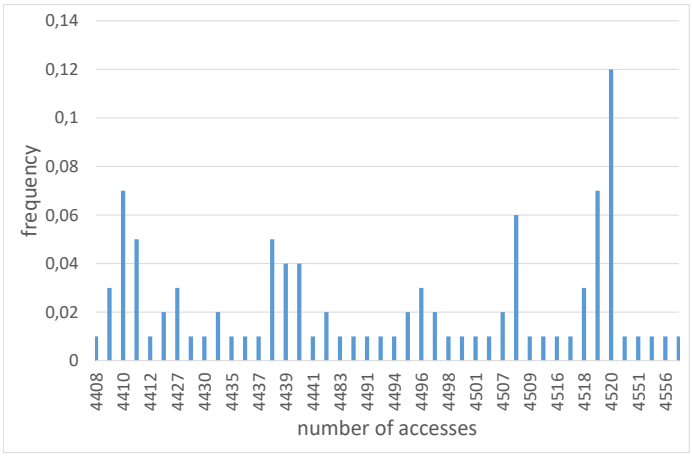


Fig. 6: Relative frequency of execution times of `ag_fast` (snapshot from RVS tool).



(a) `ag_fast`



(b) `ag_slow`

Fig. 7: Frequency of accesses to LMU0.

and tr_B is initiated by $task_B$ hosted by $core_Y$ ($X \neq Y$), Res_A is the resource that serves tr_A and Res_B is the resource that serves tr_B. To show that the interference scenario does not exist, we can show that $task_A$ is not sensitive to the service of tr_B by res_B and that $task_B$ is not sensitive to the service of tr_A by res_A.

Let us take the example of a non-interfering scenario `< rd_data_ag_fast || wr_data_ag_cpcs >`, `rd_data_ag_fast` is initiated by `ag_fast` (hosted by core 2) and targets LM0 while `wr_data_ag_cpcs` is initiated by `ag_cpcs` (hosted by core 4) and targets LMU1. The absence of interference can be shown by running `ag_fast` (resp. `ag_cpcs`) against co-runners on core 4 (resp. core 2) targeting LMU1 (resp. LMU0). The expected outcome, if the non-interfering scenario is valid, is no variation in the task execution time compared to the task behavior in isolation, as shown for `ag_fast` in Figure 8 (ag_fast vs 1 contender lmu1, the slight variation is due to code instrumentation for measurement).

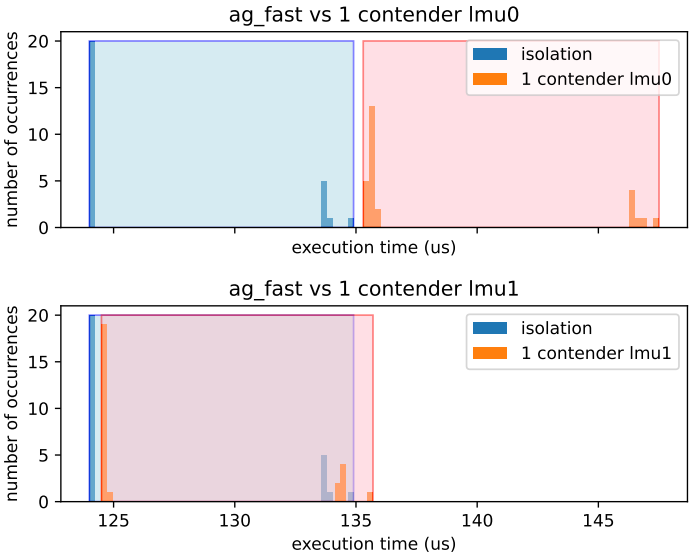


Fig. 8: Distribution of `ag_fast` execution times: in isolation (in blue) and vs contenders (in red).

D. Summary

In this section, we have shown an application of our method (introduced in Section IV) to build a PML model. Refinement has been applied on the PML model with the objective to meet the budget constraint (execution times evaluated from the model must fulfill the budgets).

The process took place in 2 phases. We firstly applied model refinement (on the structural, routing and temporal views) in order to reduce interference scenarios to be later evaluated. In the second phase, we evaluated execution times in order to check compliance with the budget constraint. The evaluation itself can be refined in order to minimize evaluation cost, e.g., we have shown two levels of evaluation based on measurement and the more precise (and costly) evaluation has only been applied when the first evaluation was not precise enough to conclude about the constraint. Validation of the model through tests has also been shown in order to confirm the modeling assumptions.

VII. Discussion

This section provides some discussion on the refinement method presented in this paper. We also identify some limitations of the method and propose possible solutions.

a) Refinement path(s): In the previous section (Section VI), we have seen an example of application of our method. Note that if model refinement and evaluation were here performed sequentially, other paths could be possible. For example, the model could have been partially refined, then evaluated to check compliance with the constraint, refined again if necessary, re-evaluated, and so on until an abstraction that meet the constraint was found. Multiple paths are possible in practice, and choosing an "optimal" refinement path is based on practice and experience.

b) Tooling: The PML model captures data from various sources. Although the models described in this paper were done "by hand", a PML model could be automatically (or at least partially) built with the data extracted from design files, source code, configurations files, etc. For example, a script has been developed in order to extract temporal exclusions from the ASTERIOS's RSF. In a complementary way, the PML analyzer has also been extended in order to calculate metrics (e.g., interferences for each component) that can be used to drive the refinement process.

c) Reachability: When building a PML model, it may be the case that reaching the solution is not possible (e.g., modeling or evaluation effort is too high) or that this solution does not even exist (because the proposed design does not meet the constraint). In these cases, it is necessary to mitigate interferences (re-design the system) and model the system again. The results provided by the PML analyzer (list of interferences, interference channels) can be used in this task and the new design elements can be incorporated in the PML model.

d) Validation: Validation of the model is an important issue. For example, in Section VI-C, we proposed to address validation through tests: tests are used to corroborate interfering/non-interfering scenarios on the real system. However, this validation strategy is biased as it only enables to validate the elements that are captured in the model: it is thus possible to converge towards an incomplete model. A way to solve this problem would be to complete the validation strategy with benchmarks specifically devised to exhibit the common micro-architectural mechanisms that may be undocumented.

VIII. Conclusion

This paper dealt with the interference problem in multicore processor embedded systems. We extended the PHYLOG approach with a method to use the PML language. This method relies on different types of refinements (structural, routing, temporal, etc.) in order to build a precise and reliable model of the system and then perform interference analysis (i.e., identify interferences and evaluate their effects) in a sound way. We showed an application of this method on an industrial use case coming from the aerospace domain.

Future work could develop validation and evaluation aspects, investigate tools to help to build PML models such as Large Language Models (LLM) to deal with large datasheets, or integrate PML with ASTERIOS or Prelude [17] development toolchains to build interference-aware real-time applications.

Acknowledgment

This work has been done as part of the ARCHEOCS project funded by the French Research Agency (ANR) and the partners of the IRT Saint-Exupéry Scientific Cooperation Foundation and from the PHYLOG 2 project funded by the France Relance program and the European Union through the NextGenerationEU program. The authors would also like to thank ASTERIOS Technologies, especially Guillaume Phavorin, and Rapita Systems for their support in the use of ASTERIOS and measurement tools.

References

[1] "General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20)," European Union Aviation Safety Agency, Tech. Rep. AMC-20, Amendment 23, 2022.

[2] W.-T. Sun, E. Jenn, H. Cassé, and T. Carle, "Automatic Identification of Timing Interferences on Multi-Core Processor in a Model-Based Approach," in *Conférence d'informatique en Parallélisme, Architecture et Système*, 2019.

[3] V. A. Nguyen, E. Jenn, W. Serwe, F. Lang, and R. Mateescu, "Using Model Checking to Identify Timing Interferences on Multicore Processors," in *Embedded Real Time Software and System Conference (ERTS'20)*, 2020.

[4] F. Boniol, Y. Bouchebaba, J. Brunel, K. Delmas, T. Loquen, A. M. Gonzalez, C. Pagetti, T. Polacsek, and N. Sensfelder, "PHYLOG Certification Methodology: a Sane Way to Embed Multi-Core Processors," in *Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[5] X. Jean, L. H. Mutuel, and R. Soulat, "Assurance of Multicore Processors: Limits on Interference Analysis," Federal Aviation Administration, Tech. Rep. DOT/FAA/TC-19/24, Mar. 2020.

[6] "Multi-Core Processors Position Paper," Certification Authorities Software Team, Tech. Rep. CAST-32A, Nov. 2016.

[7] D. Dasari and V. Nelis, "An Analysis of the Impact of Bus Contention on the WCET in Multicores," in *IEEE International Conference on High Performance Computing and Communication & IEEE International Conference on Embedded Software and Systems*, 2012.

[8] J. Bin, S. Girbal, D. Gracia Perez, A. Grasset, and A. Merigot, "Studying Co-Running Avionic Real-Time Applications on Multi-Core COTS Architectures," in *Embedded Real Time Software and System Conference (ERTS'14)*, 2014.

[9] S. Girbal, J. Bin, D. Gracia Perez, and A. Merigot, "Using Monitors to Predict Co-Running Safety-Critical Hard Real-TTime Benchmark Behavior," in *Conference on Information and Communication Technology for Embedded Systems*, 2014.

[10] V. Brindejonc and A. Roger, "Avoidance of Dysfunctional Behaviour of Complex COTS used in an Aeronautical Context," in *Congrès de Maîtrise des Risques et Sûreté de Fonctionnement*, 2014.

[11] "Infineon AURIX TC3xx User Manual Part1," infineon, Tech. Rep. (V2.0.0), 2021.

[12] V. David, A. Barbot, and D. Chabrol, "Dependable Real-Time Systems and Mixed-Criticality: Seeking Safety, Flexibility and Efficiency with ASTERIOS," Krono-Safe, Tech. Rep., 2017.

[13] V. David, J. Delcoigne, E. Leret, A. Ourghanlian, P. Hilsenkopf, and P. Paris, "Safety Properties Ensured by the OASIS Model for Safety Critical Real-Time Systems," in *Computer Safety, Reliability and Security*, 1998.

[14] F. Siron, D. Potop-Butucaru, R. de Simone, D. Chabrol, and A. Methni, "The Synchronous Logical Execution Time Paradigm," in *Embedded Real Time Software and Systems (ERTS2022)*, 2022.

[15] Mutuel, Laurence H., X. Jean, V. Brindejonc, A. Roger, T. Megel, and E. Alepins, "Assurance of Multicore Processors in Airborne Systems," Tech. Rep. DOT/FAA/TC-16/51, 2017.

[16] S. Bayless, N. Bayless, H. Hoos, and A. Hu, "SAT Modulo Monotonic Theories," in *AAAI Conference on Artificial Intelligence*, 2015.

[17] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task Implementation of Multi-periodic Synchronous Programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, Sep. 2011.