Modelling and analyzing multi-core COTS processors

Frédéric Boniol, Julien Brunel, Kevin Delmas, Claire Pagetti, Victor Jegu

To cite this version:

Frédéric Boniol, Julien Brunel, Kevin Delmas, Claire Pagetti, Victor Jegu. Modelling and analyzing multi-core COTS processors. 11th European Congress on Embedded Real Time Software and Systems (ERTS 2022), Jun 2022, Toulouse, France. hal-03761937

HAL Id: hal-03761937
https://hal.science/hal-03761937v1

Submitted on 26 Aug 2022

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

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

Modelling and analyzing multi-core COTS processors

Frederic Boniol, Julien Brunel, Kevin Delmas, Claire Pagetti
ONERA, Toulouse, France

Victor Jegu
Airbus, Toulouse, France

***Abstract*—To embed multi-core COTS processors in an avionic product, the platform must be thoroughly analyzed from two perspectives: the worst case real-time behaviours and the safety impact of internal failures. Both activities are very complex and error-prone for large size systems. Moreover, the frameworks for both perspectives (real-time and safety) are completely decoupled, leading to independent and possibly incoherent analyses.**

Our purpose is to unify both worlds and help designers in their certification process. To this end, we have formalized and unified as much as possible the different perspectives of multi-core analysis. We have also proposed a simple description language for the platform, which contains the minimal concepts needed by both perspectives, as well as an automatic translation to the two analysis frameworks.

I. INTRODUCTION

Aeronautical safety critical systems are subject to *certification*, meaning that a *certification authority* assesses the compliance of the product with a set of adequate standards.

a) ***Certification of multi-core COTS – CAST 32A:*** The CAST32-A position paper [1] provides a set of guidance for software planning and verification on multi-core-based systems. Indeed, multi-core chips, i.e., chips integrating several cores interconnected by a shared bus, face important challenges for their integration in safety critical environment. There are two main types of analysis to perform: worst case real-time analysis and safety analysis.

Real-time and interference As a matter of fact, it is very difficult to ensure *time predictability* [2], [3] for multi-core COTS, one of the key elements requested by certification. *Time predictability* is the capability to compute a safe and tight upper bound of the number of cycles required to execute a piece of software in the worst case. For multi-core COTS, the problems come from the intensive resource sharing, the lack of documentation and the complex internal behaviour (e.g. cache coherence) to increase the average performance. For mastering the worst case behaviour, the CAST32-A promotes the computation of *interferences* – situation where several applications execute in parallel and encounter a serious timing delay compared to when executing in isolation – and *interference channels* – shared resource of the platform.

Internal failures and safety effect The classical approach was to consider the processor as a whole such that any failure leads to the complete failure of the system. Such an approach is considered as a bit naive and pessimistic for multi-core. Indeed, if a core fails, the rest of the platform can still work correctly and the global system can still be safe. Thus, making a sharper analysis decreases the pessimism. On the other hand, modern processor architectures integrate many components and intelligence such that they can be seen as systems themselves. Identifying the failure modes, their effects and their failure rates is rather challenging. Some works, such as [4], propose to emulate a component failure and observe the reaction of the platform. Others, such as [5], [6], propose to deduce abstract failure modes from the functional services in pragmatic reasoning approach. Some, e.g. [7], try to quantify the failure rate with real platform experiments. For mastering the failure propagation, the CAST32-A promotes the identification of internal failures and their containment within the equipment (integrating the multi-core) not to polute the avionics.

b) ***Objectives and contribution.:*** Practically, the applicant must argue that they have identified the interference and the safety effect for their platform and their specific use. The analyses are applied on the platform which includes the hypervisor or RTOS (real-time operating system) if any. By specific use, the CAST32-A speaks of *configuration settings*, i.e. the way the processor is used. This includes the description of which components are used and how (with which parameters).

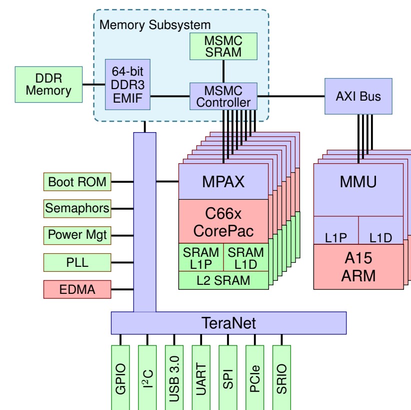

Fig. 1. KEYSTONE platform

Let us consider as an example the KEYSTONE TCI6630K2L [8] from Texas Instruments, which is depicted in Figure 1. The configuration settings include which cores are running and with which frequency; the peripherals that are used, how the memory is configured and so on.

Once the configuration settings have been clearly described, the applicant must then identify the interferences, i.e., compute

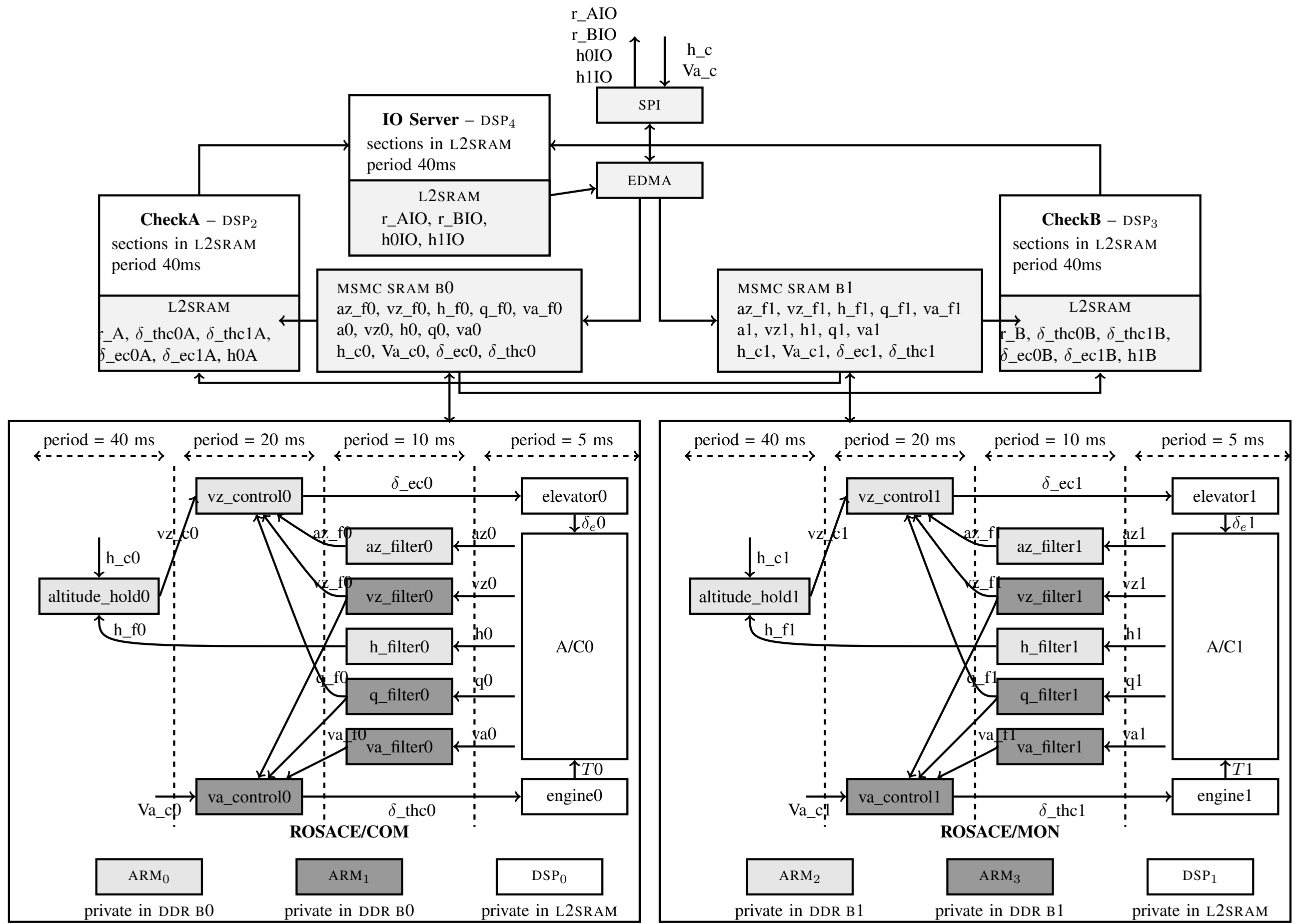

Fig. 2. Adapted RROSACE application

how software could access the different resources in parallel. The basic solution is to compute all *transactions* – accesses from a core to a shared resource – and enumerate all combinations with a solver [6].

In parallel, the applicant must also identify the failure modes of each internal component of the platform and determine how these failure modes would impact the transactions. For instance a non acceptable transaction (outside the configuration settings) can occur in the presence of some failure. The basic solution is to make a dysfunctional model and analyze the safety effect [9] with a safety framework, such as [10].

Both perspectives deal with the notion of transaction and how the platform is solicited. However, they are analyzed via independent tools and techniques. Our objective is to unify them as much as possible to factorize the modelling work and reduce the divergence between the perspectives. To do so, we have formalized the notion of transaction, interference and erroneous transaction. We have defined a multi-core-based system description framework to 1) describe thoroughly the platform with the common concepts needed by the analyses; 2) translate the description to each perspective and in a format that is compliant with the analysis tools. This framework was developed within the project PHYLOG[1].

*c) **Outline of the paper.**:* To help illustrate the concept, we have defined a complete use case based on the KEYSTONE in Section II. Section III provides the formalisation of the multi-core transactions and the common description language named PML. Section IV presents the interference analysis and how such an analysis is possible from PML. In a similar way, Section V presents the safety perspective. We then detail the related works in Section VI before concluding.

II. USE CASE

To help present the contribution, we will rely on a real use case that consists in executing a simplified longitudinal flight control system (see Figure 2) on the KEYSTONE.

*a) **KEYSTONE.**:* The platform, shown in Figure 1, runs in *bare-metal* (*i.e.* without any RTOS and hypervisor) and is composed of: 1) an eight C66 DSP pack, in which each core comes with dedicated L1 and L2 caches, and a memory extension and protection unit (MPAX); 2) a four ARM pack, in which each core comes with dedicated L1 caches, and a memory management unit (MMU); 3) a central memory

[1] https://w3.onera.fr/phylog/

system that gives access to the platform's SRAM (MSMC SRAM), and an external DDR. Each of the these two memory systems is composed of 8 Banks, which are denoted B_x in the sequel. The memory access management is performed by the Multicore Shared Memory Controller (MSMC); 4) a set of IO peripherals (e.g. GPIO, UART), and utility peripherals (e.g. Boot, Semaphores); 5) a memory transfer peripheral (EDMA); 6) an ultra speed bus (TERANET) connecting the peripherals, the memory systems, and the cores.

b) ***Applications****:* We consider a COM/MON longitudinal flight controller which is an adaptation of RROSACE (for redundant ROSACE) [11], [12]. The purpose is to execute two parallel ROSACE– an open source longitudinal flight controller – and to perform regular verification that both copies, named COM/MON for COMmand and MONitoring, agree on the computed orders. To do so, the orders are usually compared in the MON duplicate. Our purpose is slightly different from [11], which goal was to offer a safe COM/MON strategy. Instead, we want to implement a representative use case that will stress several hardware components of the multi-core. Moreover, we have embedded the aircraft models to increase the size of the footprint and to be close to the real behaviour.

The overall use case is described in Figure 2. To allow the communication with the cockpit to receive pilots orders (required altitude *h_c* and required *Va_c*), we implemented a communication with the SPI (Serial Port Interface). We use the same medium to display the results. This results in implementing 5 functions:

1) ROSACE COM which has been allocated on several cores. The environment is on DSP_0 which is configured with a L2SRAM such that the execution is contained locally, except for the data that is exchanged with the controller. Those global variables are stored in MSMC SRAM B_0. The controller has been split in two parts: one executing on ARM_0 and the second on ARM_1. All the private sections are stored in DDR B_0. The ARM caches are activated. The controller receives orders (*h_c0*, *Va_c0*) from the pilot and computes the orders *δ_ec0* and *δ_thc0*.

2) ROSACE MON works in the same way except that the aircraft model is on DSP_1, the controller is on ARM_2 and ARM_3, the private sections are on DDR B_1 and the global variables are in MSMC SRAM B_1. The controller receives the same orders as COM and they are stored as *h_c1* and *Va_c1*; and computes the actions *δ_ec1* and *δ_thc1*.

3) CheckA executes on DSP_2. Its L2 is configured as L2SRAM and contains all the sections and data. CheckA reads several data in the MSMC SRAM B0 (*δ_ec0*, *δ_thc0*, *h0*, *Va_c0*) and MSMC SRAM B1 (*δ_ec1*, *δ_thc1*, *h1*, *Va_c1*). It then checks whether the orders computed by COM and MON are close, e.g. by verifying whether $|\delta_ec0 - \delta_ec1|$ and $|\delta_thc0 - \delta_thc1|$ are small, and CheckA stores the result in the Boolean variable *r_A* (which is true if COM and MON agree, and false otherwise).

4) CheckB works as CheckA and computes *r_B*.

5) IO server executes on DSP_4. Its L2 is configured as L2SRAM and contains all the sections and data. The IO server is in charge of communicating with the outside of the multi-core via the SPI. More precisely, it configures the EDMA to receive the pilot orders from the SPI and copy them on MSMC SRAM B0 and MSMC SRAM B1. It also periodically reads the outputs of CheckA and CheckB directly in their L2SRAM and copy them locally in its L2SRAM. It configures the EDMA to send those to the SPI.

III. MODELLING MULTI-CORE ARCHITECTURE: PML

To prepare the certification documentation required by the CAST32-A, the applicants must analyze the platform from the two perspectives, real-time and safety. Even if they differ in terms of framework, they both rely on an accurate representation of the platform itself. Such a representation is derived from hardware documents and expert knowledge.

A. Components

The software are hosted by hardware components. When a software requests some resource, it initiates a *transaction* within the platform. This transaction consists of a *path* of physically connected components. According to their role in a transaction, components are classified as follows, taking inspiration from the initiator-target model introduced in [13], [14], [15], [6].

Definition 1 (Initiator-target model): A multi-core is composed of three types of components:

Initiator: a component which initiates a transaction (e.g. ARM, DSP and EDMA);

Target: an end-component which is targeted by initiators (e.g. MSMC SRAM and SPI);

Transporter: any intermediate component between initiators and targets (e.g. TERANET and AXI BUS).

Example 1: The components of the KEYSTONE illustrated in Figure 1 are colored according to their type, the color code being: red for Initiators, blue for Transporters and green for Targets.

The mapping of the (software) applications to the platform components defines the configuration of the platform and induces which components / transactions are active.

Example 2: The set of RROSACE software components and their allocation are defined in Figure 2. In particular, DSP 5-7 are turned off, several peripherals are disabled, several DDR and MSMC SRAM Banks are unused. Figure 3 shows the final configuration.

B. Transactions and services

The interaction between software and platform is abstracted away through the notion of service. Indeed, when a software initiates a transaction within the platform, e.g. to retrieve data, each component along this transaction plays its role by providing a service. Components offer many services such as execute or address translation. But in the context of this article, we focus on the minimal services that are needed for the interference and safety perspectives, *i.e.*, LOAD and STORE.

Example 3: Let us consider again the KEYSTONE with the configuration described in II. Application *az_filter_0* may

need to read data stored in the MSMC SRAM B_0 memory. This is expressed as a LOAD service call and consists in a transaction propagated through internal components until the DDR is reached. Besides, each of these components provides a LOAD service. Figure 3 shows an extract of the service-oriented KEYSTONE architecture.

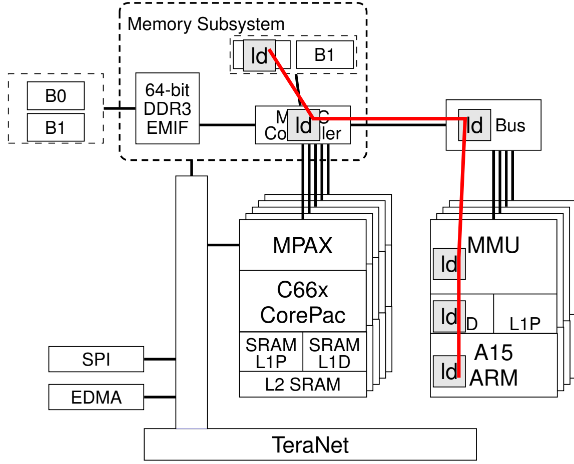

Fig. 3. Example LOAD transaction

Definition 2 (Platform service): A platform offers a set of services that can be called upon by the Initiators and that generate transactions. We have identified the following services:

- LOAD: retrieval of some data from a given target by an initiator.
- STORE: writing of some data to a given target by an initiator.

Definition 3 (Component service): We consider that all components offer both services (LOAD and STORE). In particular for a component c, we will use the notation c_l (resp. c_s) to represent the LOAD (resp. STORE) service offered by the component c. For a component service s, $cp(s)$ is the component that provides s.

Example 4: The component service ARM_0_l is provided by the component $cp(\text{ARM}_0_l)$ = ARM_0.

Definition 4 (Transaction): A transaction is initiated by an initiator and follows a pre-defined *path* to connect the initiator to the final target. We denote a LOAD (resp. STORE) transaction tr that represents the initiator i reaching the target t as $i \rightarrow^l t$ (resp. $i \rightarrow^s t$).

Remark 1: We make the distinction between LOAD and STORE transactions for two main reasons: firstly, for some platforms (not for the KEYSTONE), LOAD and STORE may use different paths; secondly they induce different effects on both interference and safety analyses. Indeed, LOAD and STORE transactions induce completely different temporal effects. Moreover, the propagation of a failure along a LOAD transaction goes from the target to the initiator whereas it goes the other way around in the case of a STORE transaction.

Remark 2: Note that the path has no specific direction. Indeed, a transaction actually represents a sequence of interactions, which can go a one way or the other. For instance, a LOAD consists in sending a request from a core to the DDR and then the data is sent back from the DDR to the core.

Remark 3: The KEYSTONE verifies the so-called *unique path property*. Indeed, any transaction $tr = i \rightarrow^X t$ with $X \in \{l, s\}$ always follows a unique path (both for resquest and data). For other types of platform, making an X from i to t could lead to several paths. In such a situation, the notation $i \rightarrow^X t$ should be enriched. In order to take this into account, we would need to introduce the concept of *single transaction*, satisfying the unique path property and we would define a transaction as a set of single transactions.

Definition 5 (Copy): DMAs (such as the EDMAs) make copies from one memory area to another one. Thus we denote the copy transaction as $\mathit{DMA} \rightarrow^{copy} [\mathit{Mem}_1, \mathit{Mem}_2]$. We can see such a transaction as the pipeline of the two transactions $\mathit{DMA} \rightarrow^l \mathit{Mem}_1$ and $\mathit{DMA} \rightarrow^s \mathit{Mem}_2$.

Definition 6 (Path): A transaction $tr = i \rightarrow^X t$ with $X \in \{l, s\}$ follows a path of components denoted by $\mathit{cp_path}(tr)$, which is a chain of *Transporters*, except the last component, which is a *Target*. Let $p = c_1 \leftrightarrow \ldots \leftrightarrow c_k$ be a component path, then two successive components c_j and c_{j+1} in p are physically connected.

Each component along $\mathit{cp_path}(tr)$ contributes to the transaction tr by providing the service X. The resulting path of services is denoted by $\mathit{path}(tr)$. Thus if $\mathit{cp_path}(i \rightarrow^X t) = c_1 \leftrightarrow \ldots \leftrightarrow c_k$, then $\mathit{path}(i \rightarrow^X t) = c_1_X \leftrightarrow \ldots \leftrightarrow c_k_X$.

Example 5 (Path): For instance, the transaction $tr_1 = \text{ARM}_0 \rightarrow^l \text{MSMC SRAM B}_0$ shown on the Figure 3 follows: $\mathit{cp_path}(tr_1) = \text{ARM}_0 \leftrightarrow \text{ARM}_0\text{_L1D} \leftrightarrow \text{MMU}_0 \leftrightarrow \text{AXI} \leftrightarrow \text{MSMC CTRL} \leftrightarrow \text{MSMC SRAM B}_0$ and $\mathit{path}(tr_1) = \text{ARM}_0\text{_l} \leftrightarrow \text{ARM}_0\text{_L1D_l} \leftrightarrow \text{MMU}_0\text{_l} \leftrightarrow \text{AXI _l} \leftrightarrow \text{MSMC CTRL_l} \leftrightarrow \text{MSMC SRAM B}_0\text{_l}$.

C. PML *metamodel*

The purpose of PML (for Multi-Core Meta-Model) is to describe the common description of a multi-core needed for both perspectives. Figure 4 provides a graphical representation of PML.

A platform is composed of several physical connected components, each with a *type* (Initiator, Transporter, Target). We represent the link between an initiator and the transactions it initiates through the association *issued by*. Each transaction tr relies on the path of services $\mathit{path}(tr)$ (association *along*) targeting a specific service among them (association *targets*). Each service (which can be of type LOAD or STORE) is provided by a component (association *provides*). Finally, instead of representing the allocation of software to hardware components, we consider an abstraction of it, simply representing the transactions that are made possible by this allocation (association from software to transaction).

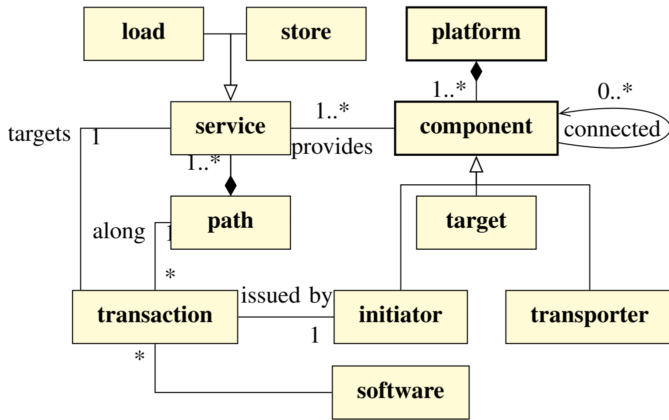

Fig. 4. Overall PML overview

D. Tooling support

The construction of an PML model is supported through a Scala API provided in the PML analyser[2]. The detailed codes of the experiments presented in this paper are provided as examples of the API usage.

The API offers a programmatic way to instantiate the physical components and applications of a platform. The description of the transactions used by the applications can be cumbersome and error-prone. Therefore, thanks to the API, one can simply specify software/hardware and data/hardware allocations from which the transactions can be automatically derived.

Validation strategy: The API also contains a set of graphical exporters that can extract specific views of the model. Such exports can be very useful during the design and the validation of the PML model. Among the possible extracts, there are:

- the physical/service connection graph of the platform;
- the physical/service connection graph restricted to the connections used by at least one transaction;
- the transactions used by a given application.

The API also provides the automatic generation of the interference model and the safety model, as detailed in the next sections.

IV. Interference analysis

One of the analyses required by the CAST32-A is the identification of all interferences and interference channels. An interference happens when two or more transactions occur simultaneously and when they use either a common service or services offered by the same component.

A. Interference calculus overview

Let us first explain what is exactly the interference calculus. The idea is to enumerate all the simultaneous transactions that can lead to a timing alteration on the application execution.

Definition 7 (Simultaneous transactions): $tr_1 || \dots || tr_j$ denotes the situation where the transactions tr_i with $i \in 1..j$ can occur simultaneously. This is only possible when the initiators are distinct: $\forall k, l \in 1..j, k \neq l \Rightarrow Initiator(tr_l) \neq Initiator(tr_k)$.

Example 6 (Simultaneous transactions): For instance, let us consider $tr_1 = \text{ARM}_0 \rightarrow^l$ MSMC SRAM B_0 and $tr_2 = \text{ARM}_2 \rightarrow^s$ MSMC SRAM B_1. tr_1 is a LOAD transaction and tr_2 is a STORE one. *Initiator*(tr_1) $=\text{ARM}_0$ and *Initiator* $(tr_2) = \text{ARM}_2$. Then we can have $tr_1 || tr_2$, *i.e.* these two transactions can be initiated simultaneously.

PML captures the minimal concepts that are needed by all analyses. When focusing on a given analysis, the corresponding view may have to be enriched. The default semantics of PML assumes that 1) two services belonging to two different components do not share any resource and thus do not interfere (*i.e.* they can simultaneously serve different transactions); 2) two services offered by the same component cannot execute in parallel since each of them needs all the resources of the component. However, some processors contain components powerful enough to provide several services at the same time. For instance, crossbars allow for parallel communications. To take this knowledge into account, we have to extend the PML model with a new relation specifying the services that can run in parallel without producing an interference.

Definition 8 (Parallel services): Let us introduce the relation *parallel*, which represents the pairs of services s_1 and s_2 that can run in parallel without conflicting on any resource.

$$(s_1, s_2) \in \textit{parallel} \iff s_1 \textit{ and } s_2 \textit{ do not interfere}$$

This input information must be given by the designer. Such a knowledge could come from a deep analysis of the processor datasheet or from precise benchmarks exploring the behaviour of each component of the plateform.

Example 7 (Parallel services): The documentation of the KEYSTONE processor states that the ultra speed bus (i.e., the TERANET component) enables LOAD and STORE transactions in parallel without any interference. For instance, if two DSPs access the DDR Banks simultaneously, one with a LOAD transaction and the other with a STORE, then they should not interfere on the TERANET. This is encoded as

$$\textit{parallel} = \{(\text{TERANET_}l, \text{TERANET_}s)\}$$

Note that for the KEYSTONE, the *parallel* relation does not include any other pair of services. This means that all the components, except the TERANET, are only able to provide one service at a time. For instance, the AXI bus cannot be simultaneously crossed by a LOAD transaction and a STORE one.

Definition 9 (Interference channel): An *interference channel* is a component c, more precisely a Transporter or a Target, such that there exist two simultaneous transactions *conflicting* on this component. By conflict, we mean that this will generate an interference and thus a timing effect.

Formally, we say that c is an interference channel iff there are two *distinct* transactions tr_1 and tr_2 such that:

[2]available at https://w3.onera.fr/phylog/

(a) either $\exists s \in path(tr_1) \cap path(tr_2)$ such that $cp(s) = c$, *i.e.* the two transactions use the same service s of component c,

(b) or $\exists s_1 \in path(tr_1)$ and $\exists s_2 \in path(tr_2)$ such that $cp(s_1) = cp(s_2) = c$ and such that $(s_1, s_2) \notin parallel$.

If one of the two conditions above holds, we say that tr_1 and tr_2 interfere and they conflict on c.

Example 8 (Interference channel): Considering $tr_1||tr_2$ of Example 6. As tr_1 is a LOAD and tr_2 is a STORE, they do not use any common service: $path(tr_1) \cap path(tr_2) = \emptyset$. However, they cross two common components: $cp_path(tr_1) \cap cp_path(tr_2) = \{$AXI, MSMC CTRL$\}$. tr_1 uses the LOAD service of AXI, *i.e.* AXI_$l \in path(tr_1)$, while AXI_$s \in path(tr_2)$.

According to Example 7, (AXI_l,AXI_s)$\notin$*parallel*. Thus, condition (b) of Definition 9 holds. tr_1 and tr_2 conflict on the AXI and similarly on MSMC CTRL.

Definition 10 (Interference): An *interference itf* is a situation where several transactions occur simultaneously and conflict on some interference channel(s), *i.e.* *itf*$= tr_1|| \ldots ||tr_n$ such that $\forall i, j \in 1..n$, if $i \neq j$ then

1) either tr_i and tr_j interfere,
2) or there exists a subset $\{tr'_1, \ldots, tr'_k\} \subseteq \{tr_1, \ldots, tr_n\}$ such that tr_i and tr'_1 interfere and $\forall l < k, tr'_l$ and tr'_{l+1} interfere and tr'_k interferes with tr_j.

We moreover denote by *trans(itf)*$=\{tr_1, \ldots, tr_n\}$ the set of transactions of *itf*. And we say that $tr_1|| \ldots ||tr_n$ is an n-*ary intererence*, or simply an n-*ary itf*.[3]

The conditions above mean that *trans(itf)* must form a connected graph (where the edges are the pairs of transactions that interfere). In other words, for each pair (tr_1, tr_2) in *trans(itf)* either tr_1 and tr_2 interfere, or there is a "path" of interfering transactions in *trans(itf)* from tr_1 to tr_2.

Example 9 (Interference): Let us consider again the transactions tr_1 and tr_2 of Example 6. $tr_1||tr_2$ is a *2-ary itf* that conflicts on AXI and MSMC CTRL.

Example 10 (Interference): Let us now consider the transactions $tr_3 =$ DSP$_3 \rightarrow^l$ MSMC SRAM B$_1$ and $tr_4 =$EDMA$\rightarrow^l$ SPI. As shown in Figure 5, tr_1, tr_2 interfere; tr_3 interferes with tr_1 and tr_2 on MSMC CTRL; tr_4, tr_3 interfere on TERANET. Thus $\{tr_1, tr_2, tr_3, tr_4\}$ is a connected graph, even if tr_4 does not interfere directly with tr_1 and tr_2. $tr_1||tr_2||tr_3||tr_4$ is then a 4-ary itf.

Definition 11 (Interference channel associated with an itf): For a given *interference itf*$= tr_1|| \ldots ||tr_n$, an interference channel associated with *itf* is an interference channel that appears between at least two transactions of *itf*. The set *chan(itf)* of the interference channels associated with *itf* is defined as follows:

$$chan(itf) = \{ \quad c \in Transporter \cup Target | \exists tr_j \neq tr_k \in trans(itf) \text{ such that } tr_j \text{ and } tr_k \text{ interfere on } c\}$$

Example 11 (Interference channel associated with an itf): As a last example, let us consider again the 4-ary itf depicted

[3]Note that we use *itf* either to denote a specific interference or to abbreviate the term "interference".

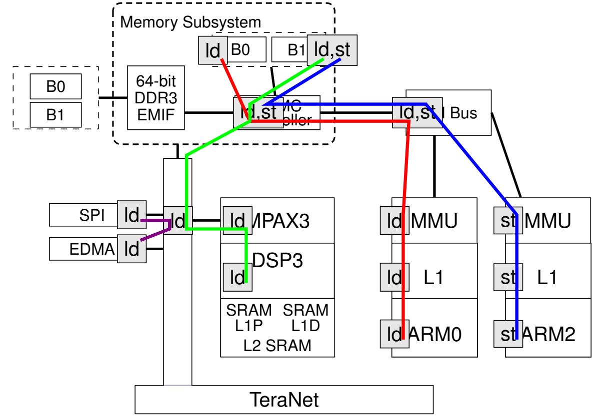

Fig. 5. Example of 4-ary itf: $tr_1||tr_2||tr_3||tr_4$

Figure 5: *itf* $= tr_1||tr_2||tr_3||tr_4$. The associated set of interference channels is *chan(itf)* $= \{$AXI,MSMC CTRL, MSMC SRAM B$_1$,TERANET$\}$.

Complementary to definition 10 which defines *n-ary itf*, i.e., a simultaneous transaction $tr_1|| \ldots ||tr_j$ which interfere by forming a connected graph, we can now define the set of interference-free simultaneous transactions :

Definition 12 (Interference-free): A simultaneous transaction $s = tr_1|| \ldots ||tr_j$ is an *n-ary interference-free* iff $\forall i, j \in 1..n$, if $i \neq j$ then tr_i and tr_j do not interfere.

As shown below (paragraph Experiments), identifying all the simultaneous transactions that are supposed to be interference-free is a way to explicit the hypotheses hidden in the model.

B. What is generated from PML?

Method 1 (Interference identification): The CAST32-A asks for the identification of all interferences. This means that we need first to determine the transactions and their path. Then, with $n = 2^{|\text{initiators}|}$ (|initiators| being the number of initiator components), we enumerate all n-*ary* itf, *i.e.* all the combination of n simultaneous transactions that may interfere. And finaly we enumarate all the associated interference channels. The way to compute the interference is then left to a solver. In our case, we use IDP [16] or MONOSAT [17]. The constraints are hard coded and are independent from the platform model. Thus, from PML, we need to generate automatically the transactions and their paths.

Example 12 (Generation of simple transactions): For the use case, the model is composed of 54 transactions, each of them being defined by its path of components. Transactions tr_i $i = 1 \ldots 4$ in Figure 5 are examples of these 54 possible transactions. This model is then enriched with the parallel relation as defined in Example 7.

C. Experiments

The interference analyser generates for each $n \in 2..N$ (where N is the number of initiators)

1) the set IF^n of n-ary interference-free simultaneous transactions;
2) the set I^n of n-ary *itf*.

Validation strategy: IF^n is interesting to check the correctness of the model. Indeed, as any combination $tr_1|| \ldots ||tr_n \in IF^n$ is supposed not to generate any interference, the idea is to measure the behaviour of $tr_1 \ldots tr_n$ in isolation and to check that it does not change when running them in parallel. The sets I^n provide the answer of the CAST32-A certification objective.

Example 13 (Interference calculus): For the use case, the interference analysis provides the following results:

Type	Size				
	2	3	4	5	6
itf	364	2 580	12 384	40 704	92 768
free	928	7 298	30 067	66 796	75 072

Type	Size				Total
	7	8	9	10	
itf	144 896	148 480	90 112	24 576	556 864
free	33 024	0	0	0	213 185

The next step after generating IF^n and I^n for all $n \in 2..N$ is to associate a benchmark $m_1|| \ldots ||m_n$ with each *itf* $= t_1|| \ldots ||t_n$ of I^n and to run it in order to quantify the interference. In the same way, a similar benchmark should be associated with each element of IF^n in order to check that there is no interference. The total number of *itf* and of interference-free simultaneous transactions seems to be too large to be tractable. However, let us note that the transactions tr_i we are considering are micro-transactions performing only one type of action (*load* or *store*). The corresponding micro-benchmarks m_i are then very small codes only repeating a same instruction a finit number of times. The typical execution time of such micro-benchmarks is about $10ms$. Running about 600 000 benchmarks would take less than 2 hours.

V. SAFETY ANALYSIS

The purpose of the safety analysis is to identify the effect of physical failures on the applications, that is the behaviour of the transactions in the presence of failures. The way to analyze the CAST32-A safety objectives to multi-core was presented at [18]. We simply sketched here the main ideas and detail the link with PML.

A. PHYLOG *safety analysis reminder*

The formal concepts of the the safety analysis are presented in details in [18]. Let us here detail and illustrate how those steps are applied in the context of multi-core-based systems. Note that to the best of our knowledge, there is no contribution to this question in the literature.

*a) **Identification of failure modes**:* As a preliminary approach, we consider two kinds of failure modes:

Erroneous The component does not properly process a transaction, which results in its corruption (data or address).

Lost The component does not process incoming transactions, which results in a deny of service.

The safety effect of these failure modes are described in the table below. Note that, the user could easily define more failure modes.

Type	FM	Comments
STORE	*err*	erroneous value or wrong destination is stored
	lost	no value is stored
LOAD	*err*	erroneous data is loaded
	lost	no data is loaded

*b) **Failure propagation model**:* The dysfunctional model describes the interconnection of physical components (e.g. cores, MMU) and the transactions that ensure the system's functions. The idea is to abstract the transactions to determine 1) whether a transaction was correctly handled, 2) what are the effects of a failure on a given transaction or 3) what are the effects of an erroneous transaction on the other transactions.

In the safety view, the path associated with a transaction tr is directed. This is due to the propagation of failures, which follows a direction along a transaction (either from the initiator to the target in the case of a STORE transaction, or the other way around, in the case of a LOAD transaction). This direction is ensured by the use of input and output ports in each component. The idea is to represent the data propagation and how their loss or corruption would affect the applications. To do that, each component is modeled as a mode automaton [19] and the whole system is the connection of all components. The behaviour of a the platform is partially illustrated in Figure 6.

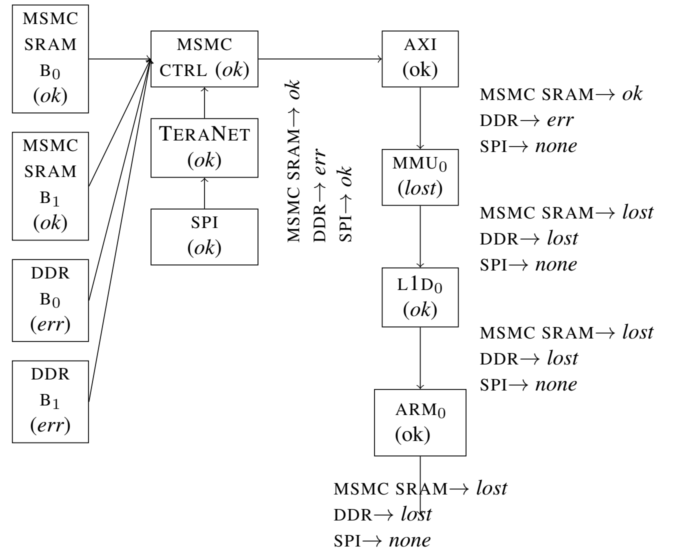

Fig. 6. LOAD transaction with some failures

Let us consider the transaction tr_1 of Figure 3, translated in a failure propagation view as shown in Figure 6. We observe the local effect of each component failure mode. In

this scenario, all components are *ok* except the MMU$_0$ which is in the *lost* mode and the DDR banks that are *err*.

The MSMC CTRL has to merge the inputs of several components (MSMC SRAM banks, DDR banks, TERANET). This merging necessitates a more complex behaviour in the mode automaton that we will not detail here. In effect, it would take, as input for Target t, the output value provided by t.

The outputs of MSMC CTRL are the inputs of AXI. The AXI BUS only considers the transactions of the ARMs.

The outputs of AXI are the inputs of MMU$_0$. Here, because of the internal failure of the component, values are not transmitted anymore to the ARM, taking then a *lost* value.

STORE works similarly with more failure propagation. First, as for the interference view, the model can be enriched with a special type of Transporters, called Virtualizers (such as the MMU or MPAX), which define the authorization for accessing transactions. To better model the effect of Virtualizer failures, we consider that an erroneous Virtualizer may access any target and pollutes these targets with *err* values. Second, a Target that receives *err* values with a STORE transaction is considered itself as corrupted from now on. This behaviour is taken into account within the hard-coded library of ALTARICA components. The interested reader can find more information on the formal encoding in [18].

*c) **Safety objective**:* In the context of our case study, the safety objective is to ensure that RROSACE controls correctly the aircraft in the following sense: if the orders sent to the actuators are *err*, this situation must be detected. The detection is done by CheckA and CheckB, thus the violation of the safety objective arises when ROSACE MON, ROSACE COM, CheckA and CheckB are *err* at the same time. Thus, the situation that we want to avoid, which is called a *failure condition* is defined by *CheckA*=err *and CheckB*=err *and COM*=err *and MON*=err.

We consider that *CheckA*=err if its L2SRAM is *err* or DSP$_2$ is *err*. Same for CheckB. *COM*=err if at least one of its LOAD is *err*. Same for MON.

B. What is generated from PML?

As for the interference view, some parts are hard coded in ALTARICA. We have developed a generic library for each type of components (Initiator, Target, Transporter). What is specific is the number of Target and which Target are visible in the set T_c (coming directly from the transactions). The global system, *i.e.* the interconnection of components, is also completely generated. Finally, the user must add its failure conditions. Those are then translated as an observer, which is the typical way to observe the output of the system. An external analyzer (CECILIA WORKSHOP [10]) is then used to perform the safety assessment out of the comprehensive ALTARICA model extracted from PML.

Validation strategy: Since the exported ALTARICA model can be imported in the CECILIA WORKSHOP, the user can benefit from:

Step-wise simulator that unfolds a failure scenario graphically to observe the error propagation encoded by the model. Such a simulator can be used to validate some well-chosen test-cases through expert consultation or fault injection if the application and the platform (or a detailed model) are available.

Sequence generator that computes all the failure scenarios up to a given size. This tool can be used to build a validation test base by computing all the scenarios containing one single failure where the feared event is observed (positive test) or where the feared event is not observed (negative test). One can then conduct a fault injection campaign based on these tests to validate both the vulnerability (with positive test) and tolerance (negative tests) of the architecture.

C. Experiments

Part of the results when assessing the failure condition is given in the Table I. Instead of directly looking at the safety objective, we just show two sub-failure conditions.

failure conditions	cut set
COM.*err*	$\{$ARM$_x$.*err* ($x \in \{0,1\}$), MMU$_x$.*err* ($x \in \{0-4\}$), MSMC SRAM B$_0$.*err*, AXI.*err*, DDR B$_0$.*err*, DSP$_0$.*err*, MPAX$_X$.*err* ($x \in \{0-4\}$), EDMA.*err* $\}$
CheckA.*err*	$\{$DSP$_2$.*err*, L2SRAM$_2$.*err*, EDMA.*err*, MPAX$_X$.*err* ($x \in \{0-4\}$) $\}$

TABLE I
SAFETY ASSESSMENT RESULTS FOR ROSACE

We observe that ROSACE COM.*err* is reached when one of the transaction has failed (ARM$_{0,1}$ $\to^l$ DDR B$_0$, MSMC SRAM B$_0$) or one of the virtualizer has failed (MMU or MPAX) or the EDMA has failed (as it could write in any target, thus in particular DDR B$_0$ and MSMC SRAM B$_0$). CheckA has failed if CheckA execution resources have failed (DSP$_2$.*err* and L2SRAM$_2$.*err*) or one of the virtualizers has failed (MPAX) or the EDMA has failed because the latter can write erroneous values in the L2SRAM$_2$.

Table II shows the timing of the framework applied on the RROSACE use case. Even if applied on a unique use case, with a quite realistic size, it shows promising scalability.

Task	Interference		ALTARICA
	IDP	MonoSat	
Model generation	4s	$< 1s$	11s
Analysis	4h45	$127s$	14s

TABLE II
TIMING PERFORMANCE OF THE FRAMEWORK

VI. RELATED WORK

Abstracting components by the services they offered is not new to analyze platform behaviour. For instance, CPA (Compositional Performance Analysis) has been widely used to compute worst case traversal time on embedded networks (such as AFDX or TSN) and the methodology considers that abstract resources provide network services [20] such as Qbv.

More recent work [21] computes memory access timing on multi-core processor with PYCPA. The multi-core itself is abstracted with its event arrival curves as a sequence of LOAD or STORE transactions (not the combination of both) and only the interaction with the memory is considered.

a) Support to design: Some works follow a different approach, without modelling the platform. This is the case for instance, of the timing analyses proposed in [22], [23], which take into account possible faults of hardware components.

For automotive system engineering, the authors of [24] have modelled the concepts that are important to the whole design process, from system engineering to software engineering. The obtained metamodel is used to ease the interaction between the different tools that are used during the development of automotive software. Although the hardware architectures are multi-core, interaction between cores is not the focus of this study.

AMALTHEA[4] is another framework proposed in the automotive domain for multi-core software development. This framework, based on the Eclipse technology, provides a metamodel for multi-core software and hardware modelling. The objective of this model-driven approach is to centralize all the information necessary for the complete development process. From this central model, it is possible to call different tools for partitioning, mapping, code generation, and trace analysis. AMALTHEA focuses on the development process and aims at reducing data exchanges between the tools involved in the process. Our approach is different since we only focus on the data necessary to the certification issues, allowing us to use a simpler metamodel for multi-core processors.

In the avionics and space fields, the DREAMS project [25] followed a similar approach by generalizing it to embedded distributed platforms, including multi-core processors. The aim of this project was to define a framework and a methodology for designing mixed-criticality systems (MCS). This framework is based on a metamodel of MCS capturing all the relevant design, implementation and configuration artefacts, and on a model-driven engineering process supported by tools focusing on design-space exploration, real-time scheduling, and reconfiguration synthesis. Thus the DREAMS framework focuses on the left branch of the V-cycle, and ranges from design model to derivation of platform configuration. Our contribution is different since we focus on the right branch of the V-cycle, and particularly on the certification activities in this branch. Our objective is not to support the design process, but to ease the generation of certification artefacts compliant with the MCP-CRI standard.

In the avionics fields, some work tried to adapt the MCP-CRI standard to COST multi-core architectures [26]. To ease design and certification stages, they propose to group the MCP-CRI objectives into three high level principles: (1) determining the final configuration, (2) managing interference channels, and (3) verifying the use of shared resources. However, they showed that predicting interference on a COTS multi-core architecture is a very challenging task because of the amount of possible scenarios. A way to overcome this difficulty is to use a formal model of the architecture and a formal analysis method to explore the set of interference channels. Such is the aim of our contribution.

Some other works have studied how to support the design of multi-core systems with the language AADL [27], [28]. The purpose it to take into account the shared resources and the software-to-hardware allocation for the analyses that come with the AADL toolset, in particular timing analyses. Comparing to our work, again, there are more details about the architecture than in an PML model, which is certification-oriented. Thus, it would be worth studying the generation of an PML model from such an AADL model. However, the information related to the load/store services that are needed by the software would need to be added to get an PML model.

b) Support to code generation: In [29], [30] the authors propose a metamodel of GPU architectures with the aim of supporting the development of application on such hardware platforms for non specialists in parallel programming. They extend the MARTE UML profile with a description of the allocation of data to memory elements. In [31], the authors extended an existing development framework dedicated to space application with the ability to handle multi-core platforms and time and space partitioning systems. This framework eases the development process by generating part of the code.

Focusing on multi- and manycore architectures, SHIM[5] (for Software-Hardware Interface for Multi-Many Core) is another framework dedicated to software design for multi- and many-core processors [32]. Its objective is to standardize the interface between the multi-core hardware and the software tools. It supports a precise description of the hardware components of the processor and its internal topology, including the processor cores, the inter-core communication channels, the routing protocols, the memory sub-system, hardware virtualization features, etc. The aim of SHIM is to provide a common metamodel enabling the use of many types of tools, including performance analysis, system configuration, auto-parallelizing compilers, and code generation. The approach of SHIM and AMALTHEA are very close in the sense they both provide a centralized model to support software design and code generation. They mainly differ by their respective application domain. AMALTHEA is promoted by automotive manufacturers for automotive systems. SHIM is developed by a consortium of multi- and manycore manufacturer for more general purpose software.

The approach of PML is similar in the way that the idea is to define a central model to be exploited in external views. However contrary to SHIM and AMALTHEA, which provide a detailed view of the architecture components, our model concentrates on an abstract definition only considering three types of component: *initiator*, *transporter* and *target*. Our claim is that such an abstraction is sufficient for interference and safety analyses.

[4]http://www.amalthea-project.org/

[5]https://www.multicore-association.org/workgroup/shim.php

VII. Conclusion and future works

We have defined a unified framework to analyze multi-core platform and partially answer the CAST32-A position paper. More specifically, we have abstracted a platform as the services it offers and modelled the interactions between software and complex hardware components via this service-based approach. The purpose was to propose a common model that covers the minimal concepts needed for both interference and safety analyses, which are required by the position paper. Thanks to this formalization, we have defined PML a metal-model dedicated to the description of any multi-core.

From such a description, we have implemented a tool that automatically generates the inputs needed for the analysis tools. For both perspectives, the approach consisted in hard-coded generic parts that can be reused for any platform and an automatic generation from the specific description of a platform. As PML encodes the minimal concepts, it is possible in each view to add more information (such as complex failure modes or failure propagation). We have run the framework on a realistic case study that was also used along the paper to illustrate the contributions.

In the future, we would like to extend PML (and the associated analyses) to consider cache coherence related behaviours. In our framework, a transaction is quite simple: it is represented by a simple sequence of connected components. With cache coherence, things become more complicated: requests can be broadcast, some transporters can initiate a transaction to send a data to another cache, etc. We also would like to better model the notion of *parallel* transactions: indeed, some components have some *capacities*, *i.e.* the ability to deal with several transactions in parallel to some extent.

Among the analyses required by the CAST32-A, there is a need to quantify the effects of interference. Thus, the applicant should define intensive benchmarking strategies [33], [34] in adequation with the interference. Thus, PML tooling should also propose an automatic translator to stressing benchmark for a given platform.

References

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

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[3] R. Wilhelm and J. Reineke, "Embedded systems: Many cores - many problems," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 2012, pp. 176–180.

[4] I. Villalta, U. Bidarte, J. Gómez-Cornejo, J. Jiménez, and J. Lázaro, "Seu emulation in industrial socs combining microprocessor and fpga," *Reliability Engineering & System Safety*, vol. 170, pp. 53–63, 2018.

[5] V.-A. Paun, B. Monsuez, and P. Baufreton, "On the determinism of multi-core processors," in *French Singaporean Workshop on Formal Methods and Applications*, 2013.

[6] L. Mutuel, X. Jean, V. Brindejonc, A. Roger, T. Megel, and E. Alepins, "Assurance of Multicore Processors in Airborne Systems," 2017.

[7] S. Houssany, N. Guibbaud, A. Bougerol, R. Leveugle, F. Miller, and N. Buard, "Microprocessor soft error rate prediction based on cache memory analysis," in *12th European Conference on Radiation Effects on Components and Systems (RADECS'11)*, 2011, pp. 412–419.

[8] Texas Instruments, "TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip," Texas Instruments Incorporated, Tech. Rep. SPRS893E, 2013.

[9] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi *et al.*, "Esacs: an integrated methodology for design and safety analysis of complex systems," in *Proc. ESREL*, 2003, pp. 237–245.

[10] *Cecilia Workshop framework*, Dassault, 2014.

[11] H. Deschamps, G. Cappello, J. Cardoso, and P. Siron, "Coincidence Problem in CPS Simulations: the R-ROSACE Case Study," in *9th European Congress Embedded Real Time Software and Systems ERTS 2018*, Jan. 2018.

[12] H. Deschamps, "Scheduling of a Cyber-Physical System Simulation," Ph.D. dissertation, Institut Supérieur de l'Aéronautique et de l'Espace, 2019.

[13] V. Brindejonc and A. Roger, "Avoidance of dysfunctional behaviour of complex cots used in an aeronautical context," in *19eme Congrès de Maîtrise des Risques et Sûreté de Fonctionnement*, 2014.

[14] X. Jean, L. Mutuel, and V. Brindejonc, "Assurance methods for cots multi-cores in avionics," in *35th Digital Avionics Systems Conference (DASC'16)*, 2016.

[15] L. Mutuel, X. Jean, and V. Brindejonc, "Investigation of error types associated with failures in multicore processors," in *20eme Congrès de Maîtrise des Risques et Sûreté de Fonctionnement*, 2016.

[16] B. de Cat, B. Bogaerts, M. Bruynooghe, and M. Denecker, "Predicate logic as a modelling language: The IDP system," *CoRR*, vol. abs/1401.6312, 2014.

[17] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "Sat modulo monotonic theories," *arXiv preprint arXiv:1406.0043*, 2014.

[18] P. Cuenot, K. Delmas, and C. Pagetti, "Multi-core processor: Stepping inside the box," in *Proceedings of 31st European Safety and Reliability Conference ESREL 2021*, 2021.

[19] A. Rauzy, "Mode automata and their compilation into fault trees," *Rel. Eng. & Sys. Safety*, vol. 78, no. 1, pp. 1–12, 2002.

[20] D. Thiele and R. Ernst, "Formal worst-case timing analysis of ethernet tsn's burst-limiting shaper," in *2016 Design, Automation & Test in Europe Conference & Exhibition, (DATE'16)*, 2016, pp. 187–192.

[21] S. Saidi and A. Syring, "Exploiting Locality for the Performance Analysis of Shared Memory Systems in MPSoCs," in *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, 2018, pp. 350–360.

[22] J. Abella, E. Quiñones, F. J. Cazorla, M. Valero, and Y. Sazeides, "Rvc-based time-predictable faulty caches for safety-critical systems," in *2011 IEEE 17th International On-Line Testing Symposium*, 2011, pp. 25–30.

[23] D. Hardy, I. Puaut, and Y. Sazeides, "Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults," in *Proceedings of the 2016 Conference on Design, Automation Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016, p. 91–96.

[24] G. Macher, E. Armengaud, E. Brenner, and C. Kreiner, "A Lightweight Meta-Model to Support Automotive Systems and Software Engineering," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

[25] S. Barner, A. Diewald, J. Migge, A. Syed, G. Fohler, M. Faugère, and D. G. Pérez, "Dreams toolchain: Model-driven engineering of mixed-criticality systems," in *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '17, 2017, p. 259–269.

[26] I. Agirre, J. Abella, M. Azkarate, and F. Cazorla, "On the Tailoring of CAST-32A Certification Guidance to Real COTS Multicore Architectures," in *12th IEEE International Symposium on Industrial Embedded Systems (SIES'17)*, 2017.

[27] J. Delange and P. H. Feiler, "Design and Analysis of Multi-Core Architecture for Cyber-Physical Systems," in *Embedded Real Time Software and Systems (ERTS2014)*, Feb. 2014.

[28] S. Rubini, P. Dissaux, and F. Singhoff, "Modeling shared-memory multiprocessor systems with AADL," in *Proceedings of the First International Workshop on Architecture Centric Virtual Integration co-located with the 17th International Conference on Model Driven Engineering Languages*

and Systems, ACVI@MoDELS 2014, Valencia, Spain, September 29, 2014, J. Delange and P. H. Feiler, Eds., 2014.

[29] A. W. De Oliveira Rodrigues, F. Guyomarc'H, and J.-L. Dekeyser, "An mde approach for automatic code generation from uml/marte to opencl," *Computing in Science Engineering*, vol. 15, no. 1, pp. 46–55, 2013.

[30] ——, "A Modeling Approach based on UML/MARTE for GPU Architecture," in *Symposium en Architectures nouvelles de machines (SympA'14)*, 2011.

[31] C. Honvault, J. Hugues, and C. Pagetti, "Model-Based Design, Analysis and Synthesis for TSP Multi-Core Space systems," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

[32] M. Kondo, F. Arakawa, and M. Edahiro, "Establishing a standard interface between multi-manycore and software tools - shim," in *2014 IEEE COOL Chips XVII*, 2014, pp. 1–3.

[33] 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.

[34] S. Girbal, J. le Rhun, and H. Saoud, "METrICS: a measurement environment for multi-core time critical systems," in *9th European Congress on Embedded Real Time Software and Systems (ERTS'18)*, 2018.