Multi-core processor: Stepping inside the box

Philippe Cuenot, Kevin Delmas, Claire Pagetti

▶ To cite this version:

Philippe Cuenot, Kevin Delmas, Claire Pagetti. Multi-core processor: Stepping inside the box. ESREL 2021, Sep 2021, Angers, France. hal-03423962

HAL Id: hal-03423962
https://hal.science/hal-03423962

Submitted on 10 Nov 2021

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.

Multi-core processor: Stepping inside the box

Philippe Cuenot

IRT St-xupry,Toulouse, France. E-mail: philippe.cuenot@irt-saintexupery.com

Kevin Delmas

DTIS, ONERA, France. E-mail: kevin.delmas@onera.fr

Claire Pagetti

DTIS, ONERA, France. E-mail: claire.pagetti@onera.fr

The last decade has seen the emergence of multi-core and many-core processors replacing historical uni-processors in most of the applicative domains. There is no doubt that the next generation of aircraft will rely on these technologies raising major issues especially with regard to safety assessment. Indeed, currently, a processor is considered as a black-box component where any single internal failure leads to the loss of all executed software. Due to the numerous resources provided by such platform, position papers like the CAST-32A promote a finer analysis of the safety impact of internal component failures. Hence there is a necessity to open the box and see such a processor as a sub-system. We introduce a formal modeling framework capturing the main characteristics of software/hardware failure propagation. This framework is applied on a simplified UAV control use-case.

Keywords: Dependability, Safety, Multi-core processor, MBSA.

1. Introduction

The design of safety critical systems often relies on the respect of norms and standards. In aeronautics for instance, the ARP (Aerospace Recommended Practice) 4754 SAE (2010) defines a safety process and the expected relations with the development process. Even if those standards may differ from one applicative domain to another, they share a common approach. This approach basically considers that a system provides functions that are implemented as a set of *items* ranging from software, hardware, sensors to hydraulic components. Thus the designer must identify the possible failures of these components and design safety mechanisms ensuring that functions are maintained even in the presence of failures.

In particular, for computer-based systems, the safety experts must analyze the effect of the failures of the processor and its peripherals on the functions. The current approach sees a processor as an atomic component that can fail. Hence any internal failure is considered as impacting the whole processor and all functions implemented by the hosted software. Such an approach is usually performed on uni-processor since most of the time an internal failure indeed leads to the complete failure and only few functions are allocated. However, with the emergence of complex multi- and many-core processors, dedicated position papers like the CAST-32A Certification Authorities Software Team (2016) promote the use of finer analyses, focusing on the safety impacts of the failures of the these internal components. For instance, if a core has failed, the others can still continue their operations and not taking this into account would degrade the opportunities offered by those highly parallelizable components.

To address this issue, the paper introduces a formal modeling and analysis framework of a multi-core processor enabling the user to automate the safety analysis of internal failures. The paper is organized as follows: the section 2 provides the state-of-the-art failure mode and propagation identification methods as well as the main families of fault-tolerance mechanisms used to handle the internal failures of multi-core processors. The necessity to open the box and see the multi-core processor as a sub-system is exposed in the section 3. The section 4 proposes a generic and modular framework to model and analyze any multi-core. Eventually, the section 5 illustrates the framework on a UAV case study.

2. Related-work

As identified by the CAST-32A position paper Certification Authorities Software Team (2016), demonstrating that the safety consequences of the internal failures of a multi-core processor are contained within the equipment (so-called failure containment) is a major issue. Thus the authors of Certification Authorities Software Team (2016) propose to analyze the interactions between inter-

nal components.

2.1. *Failure modes identification*

The identification of the failure modes of the internal components can be classically performed by a Failure Mode Effect Analysis (FMEA). This analysis identifies the effects of the failure modes of the low-level components (for instance listed in Rebaudengo and Reorda (1999)) on higher level components or systems. Up to now, there exist very few FMEA identifying the failure modes for a given platform. It can be explained by the in-depth architecture knowledge needed to perform an FMEA and by the reluctance of chip makers to (legally) commit themselves to provide a detailed FMEA.

Methods like Villalta et al. (2018) propose to emulate a component failure and observe the reaction of the platform. Nevertheless, the quality of the results highly depends on the capacity to ensure the completeness of such approach.

In this paper, the identification is based on the approach proposed by the authors of Paun et al. (2013); Mutuel et al. (2017) that is a deduction of abstract failure modes from the functional services *i.e* a failure mode encodes the way a component fails to provide some services.

2.2. *Failure propagation*

Most of approaches like Brindejonc and Roger (2014); Jean et al. (2012) identify the data-paths where collisions may occur, called *interferences*, and analyze their safety consequences. The authors of Brindejonc and Roger (2014) base their identification on a *brute-force* enumeration of (*initiator*, *target*) combinations; whereas the authors of Bieber et al. (2018) use a more compact enumeration based on clustered interferences called *interference classes*. Our framework is clearly inspired by the notion of interference but adapts it for failure propagation analysis.

Standard formalisms like fault trees or Markov chains described in Villemeur (1992) could be used, but Model-Based Safety Assessment (MBSA) presented in Bozzano et al. (2003) has emerged as an alternative to these formalisms since it offers better modeling opportunities (see Dassault Falcon 7X Rauzy et al. (2007) for instance). ALTARICA introduced by Arnold et al. (1999); Prosvirnova (2014) is one of the most famous MBSA language whose semantics defined in Rauzy (2002) is founded on the *mode automata*. Numerous industrial-used mod- elers and analyzers (like Dassault (2014) (2014)) are based on ALTARICA. These are the reason why we chose AltaRica and mode automata to develop our framework.

2.3. *Fault-tolerance mechanisms*

Eventually, the designer must integrate safety mechanisms to ensure fault-tolerance to face internal hardware failures. Multi-core processors are composed of several complex components, among which the hardware memory protection the so-called Memory Management Units (MMU) and Peripheral Access Management Units (PAMU). As identified by Paun et al. (2013), MMUs and PAMUs failures may have a huge impact on the functions and as such must be carefully modeled and analyzed. That is why the modeling of MMUs and PAMUs was closely addressed both in the framework formalization and in the use-case.

Most of the failure containment approaches rely on *an executive layer i.e* a piece of software managing the applications access to the multi-core processor resources. A popular example of this strategy is the Integrated Modular Avionic platforms where a processor hosts several *partitions* (sets of software) for which the executive layer must prevent any interference. The framework integrates these segregation insurances as modeling assumptions (*e.g.* simultaneous transactions on the interconnect has no safety effects).

Approaches like Esposito et al. (2017); Esposito and Violante (2017) use the massive resource of the multi-core processor to implement software-based fault-tolerance patterns. These patterns do not impact the way failures spread in the multi-core processor but enhance the software resilience to platform service failures. In our framework, such software resilience is handled through dependency modeling.

3. General overview of the approach

Given a system and some *failure conditions* (combinations of functional failures leading to potential severe outcomes), the safety experts must verify that an *architecture* – *i.e* an interconnection of physical components (e.g. processors, sensors, actuators) and software components (e.g. application, hypervisor) that ensure the system's functions – fulfills the safety objectives. This is achieved as follows: 1) identify the failure modes of each component, 2) model the failure propagation within the system, 3) and assess the safety objectives.

The usual approaches to analyze the safety impacts of hardware failures are depicted by the figure 1. Each function is associated with the *physical resources* (software and hardware items) that implement it. This association encodes the *dependencies*, *i.e* the minimal sets of items needed by the function to be performed properly. In particular, a uni-processor is represented as one hardware item and is considered as a dependency of a function if some software implementing it is executed on the uni-processor.

As promoted by Certification Authorities Soft-

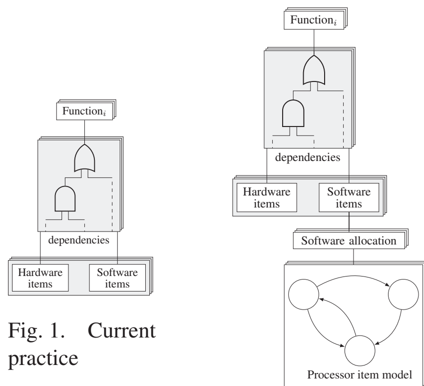

Fig. 1. Current practice

Fig. 2. Adapted practice

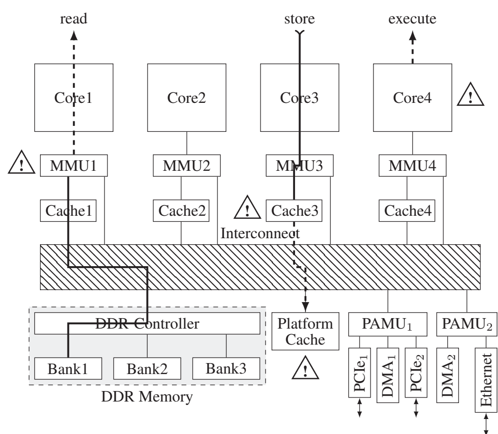

Fig. 3. Abstract representation of the NXP T1042

ware Team (2016), we propose to further refine the model of a processor with the approach of figure 2 where the processor is modeled as a dynamic system. This system is composed of several elements and software dependencies with the processor item are expressed via an *allocation*.

3.1. *Service-based modeling approach*

We consider the *first level* design as described in most processors data-sheets, such as the NXP T1042 Freescale (2016) design of the figure 3, where atomic components are the cores, the caches, the interconnect and peripherals.

In addition to the hardware components, we also consider that the platform provides high level services to the software. Part of those services can be implemented with an execution layer. In any case, the platform is configured in a certain way and this configuration provides an *allocation i.e* a description of how the software items are mapped on the processor and how the services are used by the software. We abstract the platform as a provider of four generic services, similar to those identified in Mutuel et al. (2017):

- **execute**: executes a piece of software on some core. For instance, in the figure 3, Core4 offers this service to any software executing on it;
- **load**: retrieves some data from a location and transfer it to some core. For instance in figure 3, Core1 asks to load a data stored in Bank1;
- **store**: writes some data to a given location from some core. For instance, Core3 asks to store a value in the *Platform Cache*.
- **copy**: transfers data from one location to another.

Based on these services, the authors of Brindejonc and Roger (2014); Bieber et al. (2018) have also distinguished different kinds of components:

- **Initiators**: smart initiators *execute* software and initiate *store/load* transactions (*i.e* cores). Non smart initiators initiate *copy* transactions (*e.g.* DMA).
- **Targets** process *store*, *load* and *copy* transactions (*e.g.* Bank, PCIe).
- **Transporters** route *store*, *load* and *copy* transactions (*e.g.* interconnect, MMU).

We will consider in the remainder of this paper that a *copy* can be seen as series of *loads* and *stores*. Note that an *execute* does not interact with any other resource than the local core; whereas *load/store* generate *transactions* linking initiators to targets.

Definition 3.1 (Transaction). *A transaction starts from an initiator and follows pre-defined* path(s). *A* load *transaction follows a path from the initiator to the target and the data comes back to the initiator. We assume for simplifying the model and due to lack of space that the paths used for the request and the data are the same. A* store *transaction follows a path from the initiator to the target (for the request and the data).*

Example 3.1. The *load* of Core1 follows the path *Core1* $\rightarrow$ *MMU1* $\rightarrow$ *Cache1* $\rightarrow$ *Interconnect* $\rightarrow$ *DDR Controller* $\rightarrow$ *Bank1*.

3.2. *Failure modes of atomic components and services*

As stated in the section 2, failure modes are derived from pragmatic reasoning. In the remainder of the paper we consider: 1) *err*: the component

Family	FM	Comments	Concrete FM
Core	err	mis-execution and corruption of load/store transactions	Register corruption by SEU
	$lost$	no software execution	OPCODE corruption
Memory	err	load/store a corrupted data	Memory area corruption
	$lost$	no load/store service	DDR controller stalled
Interconnect	err	corruption of the transactions	Internal queue corruption
	$lost$	transactions are not dispatched	Internal queue overflow

Fig. 4. Atomic component failure modes

does not provide a proper service, 2) *lost*: the component does not provide the service.

These abstract failure modes and the link with concrete failure modes (partially extracted from ESA-ESTEC Requirements & Standards Division (2009)) are summarized in the table 4.

Functions are affected by the *load/store* transactions failures, which result from atomic component failures. For instance in the figure 3, if Core1 asks to load a data from Bank1 and if MMU1 is *lost*, the *load* service to Bank1 is no more provided to Core1. Similarly, if Core3 asks to *store* a value in the Platform Cache and if Cache3 is *err* then an erroneous value is stored in the Platform Cache, corrupting its content. The table 5 details the service failure modes.

4. Modeling framework

The framework is a library of reusable components formalized with the *mode automata* Rauzy (2002). Such a formalization enables us to: 1) be generic and formal, 2) implement the following definitions as a library of components coded with ALTARICA, 3) use the existing tools to perform automatic safety assessments.

4.1. *Failure modes and interface*

The dynamic dysfunctional behavior of each atomic component (initiator, target and transporter) is encoded as a mode automaton. To encode the failure propagation, each component input (resp. output) represents the current status of a transaction. The failure modes are given as a set $\mathcal{F}$ including ok.

Example 4.1. *The abstract failure modes of section 3.2 are* $\mathcal{F}_{ole} = \{ok, err, lost\}$

Type	FM	Comments
store	err	erroneous value or wrong destination store
	$lost$	no value stored
load	err	erroneous data load
	$lost$	no data load
execute	err	mis-execution of the software
	$lost$	no software execution

Fig. 5. Services failures

Definition 4.1 (Transactions). *Let* $\mathcal{T}$ *be the set of targets, then a (*load/store*) transaction is a partial function* $r : \mathcal{T} \to \mathcal{F}$ *that gives the observed service failure modes for a subset of targets.*

In the sequel, $\mathcal{R}_{\mathcal{F}}$ denotes the set of partial functions $\mathcal{T} \to \mathcal{F}$. Let $f : X \to Y$ be a partial function, $expr_i$ be the expression producing some $y \in Y$ when x satisfies $cond_i$ then f is defined as follows:

$$f(x) = \begin{cases} expr_1 \text{ if } cond_1 \\ \cdots \\ expr_n \text{ if } cond_n \end{cases}$$

Example 4.2. *For the T1042 of figure 3 we have* $\mathcal{T} = \{$*Bank*$_1$*(*B_1*), Bank*$_2$*(*B_2*), Bank*$_3$*(*B_3*), Platform Cache(PC), PCIe*$_1$*, PCIe*$_2$*, Ethernet*$\}$.

Let us consider the load *of a data from Bank*$_1$ *by Core*$_1$*. The transaction follows the path* Core1 → MMU1 → Cache1 → Interconnect → DDR Controller → Bank1. *If all atomic components are* ok*, the transaction is* $r(x) = ok$ *if* $x = B_1$*. However if, as shown by the figure 6, the MMU*$_1$ *fails, no value is available, hence the transaction is* $r(x) = lost$ *if* $x = B_1$

The store *transaction, when Core3 asks to store a value in the Platform Cache and when everything is ok, is* $r(x) = ok$ *if* $x = PC$*. If the* Interconnect *is erroneous, it becomes* $r(x) = err$ *if* $x = PC$*. By doing so, the atomic component* Platform Cache *switches its internal failure mode to* err*. The corruption of* Platform Cache *is*

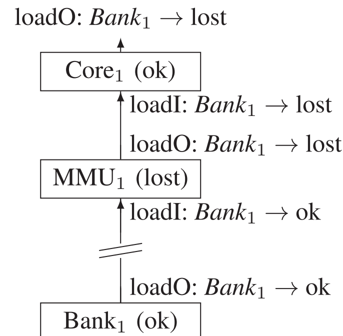

Fig. 6. Failed *load* transactions

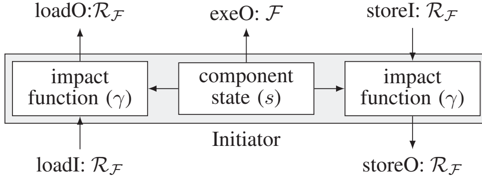

Fig. 7. Initiator component

innocuous until a core performs a load *resulting in an erroneous data.*

4.2. *Initiator*

Due to lack of space, we only show the case of a smart initiator (see figure 7). The inputs and outputs associated to the mode automaton are: 1) *exeO* provides the failure mode of the initiator; 2) *loadI* provides the result of all possible *load* transactions seen by the initiator; 3) *loadO* is the resulting *load* transactions provided to the software; 4) *storeI* provides the list of the targets the software wishes to write to; 5) *storeO* is *store* transactions sent by the initiator.

As stated before, each atomic component is modeled with a mode automaton Rauzy (2002) introduced by the definition4.2.

Definition 4.2 (Mode automaton). *A mode automaton is a 8-uplet $\mathcal{M} = \langle \mathcal{E}, \mathcal{I}, \mathcal{O}, \mathcal{S}, s_I, \delta, \sigma, dom \rangle$ where :*

- *$\mathcal{E}$ is a finite set of events;*
- *$\mathcal{I}$ (resp. $\mathcal{O}$) is the set of input (resp. output) identifiers;*
- *$\mathcal{S}$ is the finite set of states and $s_I \in \mathcal{S}$ is the initial state;*
- *dom is the domain function associating to each input and output its possible values. Let $X \subset \mathcal{I} \cup \mathcal{O}$, we introduce $dom(X) = \prod_{x \in X} dom(x)$ the Cartesian product of the domains of X. A valuation $V \in dom(X)$ associates a value to each element in X and we denote $V[x]$ the value of x;*
- *δ is a partial function called transition function $dom(\mathcal{I}) \times \mathcal{S} \times \mathcal{E} \to \mathcal{S}$ giving the next state reached from a given state and a given valuation of inputs when an event occurs;*
- *σ is a total function called output function $dom(\mathcal{I}) \times \mathcal{S} \to dom(\mathcal{O})$ giving the valuation of output according to a given state and valuation of inputs.*

Definition 4.3 (Initiator). *The initiator is modeled with a mode automaton where $\mathcal{I}$, $\mathcal{O}$, dom and σ are defined as shown by the figure 7; $\mathcal{E} = \{e_m\}_{m \in \mathcal{F}}$ where e_m is the event associated to the failure mode m; $\mathcal{S} = \mathcal{F}$ and the initial state is $s_I = ok$; γ is an impact function $\gamma : \mathcal{R}_\mathcal{F} \times \mathcal{F} \to \mathcal{R}_\mathcal{F}$ that encodes the effect of the initiator's failure mode on the* load/store *services; $\mathcal{F}, \delta$ and γ must be defined by the user.*

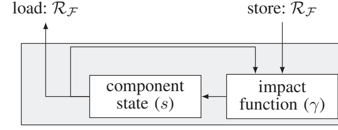

Fig. 8. Target component

Example 4.3 (Core). *Let us consider a core with the abstract failure mode $\mathcal{F}_{ole}$ and the following transition δ and impact γ functions:*

$$\delta(V, s, e_m) = \begin{cases} lost \text{ if } s \neq lost \wedge m = lost \\ err \text{ if } s = ok \wedge m = err \end{cases}$$

$$\gamma(r, s) = \begin{cases} r_3 \text{ if } s = err \\ r_2 \text{ if } s = lost \\ r \text{ otherwise} \end{cases}$$

Let $V[storeI] = r$ and $r(x) = ok$ if $x = PC$, then according to the definition of the core: 1) If the core is ok, *then $V[storeO] = r$. 2) If the core is* lost, *then $V[storeO] = r_2$ where $r_2(x) = lost$ if $x = PC$. 3) If the core is* err, *then $V[storeO] = r_3$ where $r_3(x) = err$ if $x = PC$.*

4.3. *Target*

The target is the final destination of a *store* and the first element of a *load*. A target can be affected by the reception of an erroneous *store*, thus an instantaneous event ε is needed to model the change of the target's failure mode.

Definition 4.4 (Target). *A target component $t \in \mathcal{T}$ is modeled as a mode automaton where $\mathcal{I}$, $\mathcal{O}$, dom are defined as shown by the figure 8; $\mathcal{E} = \{e_m\}_{m \in \mathcal{F}} \cup \{\varepsilon\}$; $\mathcal{S} = \mathcal{F}$ and the initial state is $s_I = ok$; $\sigma(V, s) = load \mapsto r$ where $r(x) = s$* ***if*** *$x = t$; $\mathcal{F}$ and δ must be defined by the user.*

Example 4.4 (Memory). *Let B be a memory bank with the abstract failure mode $\mathcal{F}_{ole}$, let r be the incoming store transaction $V[store]$. Let us suppose that an erroneous* store *modifies the state s of B from ok to err and conversely that a correct* store *restores the state from err to ok. The transition function δ is:*

$$\delta(V, s, e) = \begin{cases} err \text{ if } e = e_{err} \wedge s = ok \\ lost \text{ if } e = e_{lost} \wedge s \neq lost \\ err \text{ if } e = \varepsilon \wedge s = ok \\ \quad \wedge r(B) = err \\ ok \text{ if } e = \varepsilon \wedge s = err \\ \quad \wedge r(B) = ok \end{cases}$$

4.4. *Transporter*

The intermediate components, called transporters, route the service transactions. Their impact is

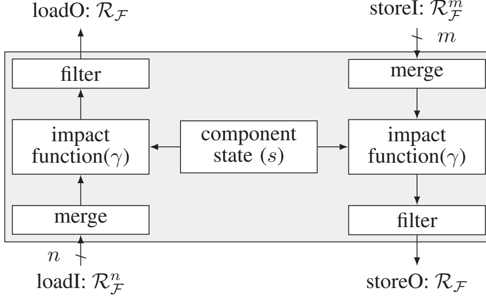

Fig. 9. Transport component

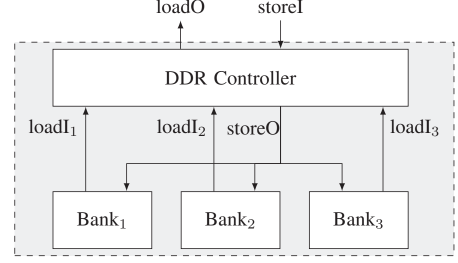

Fig. 10. DDR model

threefold: 1) change the failure mode of a transaction due to a local failure mode; 2) merge the received transactions; 3) filter the received transactions to ensure partitioning.

The transporter's interface (see figure 9) is composed of 1) $loadI$ is a set of *load* transactions; 2) $loadO$ is the result of the merging and filtering of $loadI$; 3) $storeI$ is a set of *store* transactions; 4) $storeO$ is the result of the merging and filtering of $storeI$.

Definition 4.5 (Transporter). *A transporter is modeled with a mode automaton where $\mathcal{I}$, $\mathcal{O}$, dom and σ are defined as shown by the figure 9; $\mathcal{E} = \{e_m\}_{m \in \mathcal{F}}$ where e_m is the event associated to the failure mode m; $\mathcal{S} = \mathcal{F}$ and the initial state is $s_I = ok$; transaction merging denoted $merge$, filtering denoted $filter$ and γ are user defined.*

Example 4.5 (Interconnect). *Let N be an interconnect with the abstract failure mode $\mathcal{F}_{ole}$ that receives* store *transactions from two sources and* load *transactions from two targets. Then, the mode automaton of N is the one of the definition 4.5 where δ and γ are defined in example 4.3; filter accepts all targets; $merge$ takes for each target the worst mode among the possible ones according to the order $<$ such that $ok < lost < err$. The definition of $merge$ and filter are:*

$$\begin{aligned} filter(r) &= r \\ merge_<(f_1, f_2) &= (x \rightarrow max_<(f_1(x), f_2(x))) \end{aligned}$$

4.5. *Platform modeling*

Since each atomic component is a mode automaton, the platform itself is the *parallel composition* (denoted $\|$) and the *connection* of those components. The definitions of these operations can be found in Rauzy (2002). By default the failure events of each component are considered as independent. Such an assumption may not be justified for some of the internal components, for instance due to their location on the chip. That is why common cause failure events can be defined to model such dependencies. Thus the user of the modeling framework should motivate the independence assumptions to ensure the representativeness of the analysis results.

Example 4.6 (DDR Memory). *Let us illustrate the case of DDR Memory composed of three banks. Its mode automaton is DDR $= B_1 \| B_2 \| B_3 \|$DDRController depicted on the figure 10 where the connections are the arrows between atomic components; B_1, B_2, B_3 are three targets as described in the example 4.4 and DDRController is a transporter derived from the example 4.5.*

5. Experiments

The use case is a quad-copter evolving within a predefined zone. The UAV is composed of two functions: 1) FDIR computing & processing safety actions *i.e* cut the power of motors; 2) TRAJECTORYCONTROL controlling the drone trajectory while remaining in the predefined zone.

The failure condition FC1 defined below could lead to a potential in-flight or on-ground collision. According to the CAST-32A, the safety effects of the internal failures of the multi-core processor must be contained within its equipment. So we must demonstrate that no single internal failure of the multi-core processor leads to FC1 and that the failure is handled within the equipment.

FC1 : TRAJECTORYCONTROL.*fail* $\wedge$ FDIR.*fail*

5.1. *Platform modelling*

The designed physical architecture, given by the figure 11, is composed of the following items: TRAJECTORYCONTROL is implemented by S_{NAV}; FDIR is implemented by three software: S_{MON} monitoring drone's state, S_{REC} computing the safety actions to perform and S_{FTS} emitting a signal to the switch to prevent cut-off when not asked by S_{REC}; the *Motors* (piloted by PCIe_1); the *Sensors* (acquired by PCIe_1); and an equipment E containing the *Switch* (piloted by PCIe_2) cutting off the power if no signal is

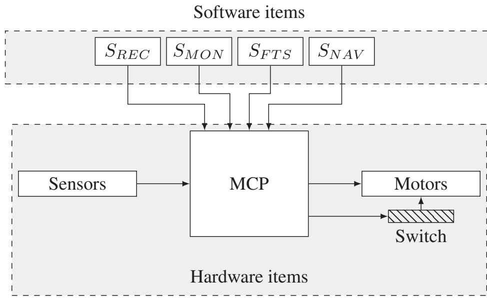

Fig. 11. RPAS Physical architecture

Software	*load*	*store*	*execute*
S_{FTS}	Platform Cache	Platform Cache	Core1
S_{MON}	Bank_3	Bank_3	Core2
S_{NAV}	Bank_1	Bank_2	Core3
S_{REC}	Bank_3	Platform Cache	Core4

Fig. 12. Service usage

received from S_{FTS} and the NXP T1042 used to execute software.

The dependencies between functions and physical items are as follows: FDIR needs that sensors and switch work properly and that S_{FTS},S_{MON} and S_{REC} are not erroneous. TRAJECTORYCONTROL relies on S_{NAV}, sensors and motors.

The dependencies between software and services are shown in the table 12. S_{FTS} *load* cut-off order from the *Platform Cache* (called PC). Let us assume that S_{FTS} is lost if the *execute* service is lost. S_{FTS} is erroneous (unintended signal to the switch) if the load value of platform cache is corrupted or if the *execute* service is erroneous. Concerning S_{NAV}, S_{MON} and S_{REC}, if the *load* service is erroneous or if the *execute* service is erroneous then the software stores an erroneous data. If the *load* or *execute* service is lost then the software does not store any data.

To provide some space partitioning, the targets reachable by each MMU/PAMU are only those the hosted software is expected to access. PCIe devices are only accessible through DMA transfer, so DMA_1 copies PCIe_1 to B_1 and B_2, B_2 to PCIe_1, and PC to PCIe_2.

We assume that: 1) the sensors, switch and motors have been selected for their high availability and integrity thus their failures are not considered in the assessment, 2) PAMU_1, Interconnect, Bank_1, Bank_2, Bank_3, PCIe_1, PCIe_2, Platform Cache, DMA_1 are protected by error correction code, so we consider only *loss* failures, but they can still be corrupted by an erroneous *store*. 3) the other caches, the Ethernet controller, DMA_2 and PAMU_2 are deactivated.

5.2. *Safety objective assessment*

Thanks to CECILIA-OCAS Dassault (2014), we computed automatically the smallest combinations of physical failures (so-called minimal cutsets) leading to FC1. If the multi-core processor is considered as a black box, then a *fail-erroneous* triggers both an erroneous trajectory and switch management (since monitoring software are erroneous), so the system does not fulfill the failure containment requirement. But when considering the refined model, the results (table 13) show that single internal failures of the multi-core processor do not trigger FC1. Moreover, the safety actions are performed by the switch, so the safety effect of any internal failure is contained within the equipment E. It may be surprising since some internal components, like the Interconnect, are involved in the processing of all load/store services. So let us explain the absence of single point of failure. To trigger the FC1, TRAJECTORYCONTROL and FDIR must fail, so 1) one of the services requested by S_{NAV} must fail; 2) one of the services requested by S_{FTS}, S_{MON} or S_{REC} must fail erroneous. Note that a loss of S_{MON}, S_{REC} or S_{FTS} does not lead to FC1 since the switch does not receive signal from S_{FTS} and cuts off motors.

The analyzer proved that a single failure among the considered internal failures cannot trigger both a failure (lost or erroneous) of the services requested by S_{NAV} and a fail-erroneous of the services requested by S_{FTS}, S_{MON} or S_{REC}. This results from 1) the resource partitioning between S_{NAV} and the other software (memories, cores, MMU), 2) the fault containment ensured by MMU and PAMU, 3) the loss-tolerance of FDIR, 4) the the absence of *fail-erroneous* for resource shared by all software (interconnect, DMA, ...).

Minimal cutsets	Effect			
	S_{NAV}	S_{FTS}	S_{REC}	S_{MON}
{C1.err, B1.lost} {C1.err, MMU3.lost} {C1.err, B2.lost} {C1.err, C3.lost} {C1.err, PCIe1.lost}	lost	err	none	none
{C4.err, B1.lost} {C4.err, MMU3.lost} {C4.err, B2.lost} {C4.err, C3.lost} {C4.err, PCIe1.lost}	lost	err	err	none
{C2.err, B1.lost} {C2.err, MMU3.lost} {C2.err, B2.lost} {C2.err, C3.lost} {C2.err, PCIe1.lost}	lost	err	err	err
{C1.err, C3.err}	err	err	none	none
{C3.err, C4.err}	err	err	err	none
{C2.err, C3.err}	err	err	err	err

Fig. 13. Minimal cutsets of FC1

6. Conclusion

Multi-core processors will be used in the next generation of safety-critical systems and before being embedded must be analyzed. We have introduced a modeling framework offering a comprehensive and generic description of most of the multi-core processor internal components. The proposed framework has been formally defined using mode automata to benefit from off-the-selves modelers and analyzers. We illustrated the framework on a remotely piloted aircraft system integrating the NXP T1042 and demonstrated how the framework can support safety assessment such as the demonstration of fault containment.

The proposed framework does not benefit from the last state-of-the-art version of the Altarica 3.0 language providing useful structural and behavioral modeling features. This choice was motivated by the availability of a graphical representation of the model and its simulation on older version of Altarica. Nevertheless the formalization provided here can be easily transposed to Altarica 3.0. On the modeling aspect, the framework does not handle some classic dynamic features like interruptions and reset procedures that might be needed to model some reconfiguration mechanisms. The introduction of modeling patterns encoding these mechanisms will be addressed by future work. Furthermore, we do not addressed the problematic of instantaneous events conflict *i.e* when two instantaneous events must be triggered at the same logical time-step. To solve this problem, we are currently developing framework features enabling the analyst to define an event scheduler.

References

Arnold, A., G. Point, A. Griffault, and A. Rauzy (1999). The altarica formalism for describing concurrent systems. *Fundamanta Informaticae 40*(2-3), 109–124.

Bieber, P., F. Boniol, Y. Bouchebaba, J. Brunel, C. Pagetti, O. Poitou, T. Polacsek, L. Santinelli, and N. Sensfelder (2018). A model-based certification approach for multi/many-core embedded systems. In *ERTS 2018*.

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

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

Certification Authorities Software Team (2016, November). Multi-core Processors - Position Paper. Technical Report CAST 32-A.

Dassault (2014). *Cecilia OCAS framework*. Dassault.

ESA-ESTEC Requirements & Standards Division (2009, March). *Space product assurance : Failure modes, effects (and criticality) analysis (FMEA/FMECA)*. Noordwijk, The Netherlands: ESA-ESTEC Requirements & Standards Division.

Esposito, S. and M. Violante (2017). On the consolidation of mixed criticalities applications on multicore architectures. *Journal of Electronic Testing 33*(1), 65–76.

Esposito, S., M. Violante, M. Sozzi, M. Terrone, and M. Traversone (2017). A novel method for online detection of faults affecting execution-time in multicore-based systems. *ACM Transactions on Embedded Computing Systems (TECS) 16*(4), 94.

Freescale (2016). T1042/T1022 QorIQ Integrated Multicore Communications Processor Preliminary Datasheet DS1176.

Jean, X., D. Faura, M. Gatti, L. Pautet, and T. Robert (2012). Ensuring robust partitioning in multicore platforms for ima systems. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pp. 7A4–1. IEEE.

Mutuel, L., X. Jean, V. Brindejonc, A. Roger, T. Megel, and E. Alepins (2017). Assurance of Multicore Processors in Airborne Systems.

Paun, V.-A., B. Monsuez, and P. Baufreton (2013). On the determinism of multi-core processors. In *French Singaporean Workshop on Formal Methods and Applications*.

Prosvirnova, T. (2014). *AltaRica 3.0: a Model-Based approach for Safety Analyses*. Ph. D. thesis, Ecole Polytechnique.

Rauzy, A. (2002). Mode automata and their compilation into fault trees. *Rel. Eng. & Sys. Safety 78*(1), 1–12.

Rauzy, A., J. Gauthier, and X. Leduc (2007). Assessment of large automatically generated fault trees by means of binary decision diagrams. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability 221*(2), 95–105.

Rebaudengo, M. and M. S. Reorda (1999). Evaluating the fault tolerance capabilities of embedded systems via bdm. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pp. 452–457. IEEE.

SAE (2010). Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems.

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

Villemeur, A. (1992). *Reliability, availability, maintainability and safety assessment*. John Wiley & Sons.