Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

System Based Interference Analysis in Capella

Amin Oueslati[a] Philippe Cuenot[ac] Julien Deantoni[b]
Christophe Moreno[ad]

a. IRT Saint Exupery, Sophia Antipolis, France
b. Université Cote d'Azur, I3S/INRIA Kairos, Sophia Antipolis, France
c. Seconded from Continental Automotive France
d. Seconded from Thales Alenia Space

Abstract In embedded systems the emergence of System on Chip (SoC) offers low cost, flexible and powerful computing architectures. These new COTS capabilities enable new applications in aerospace domain with more integration of avionic functionalities on a same hardware. The main drawback of such integration is the difficulty of mastering application's deployment on SoC architecture, while understanding miscellaneous emerging behaviors. Model Based Engineering techniques have been introduced to assist in system analysis at early stages of development process. For instance, Capella [BVNE] is a tooled language to support design of systems architecture (`http://polarsys.org/capella`). Capella helps to provide a consistent view of system architecture.

However, Capella does is not satisfactory to understand emerging behaviors. For instance it is not useful to understand how deployment of different tasks (and their parameters) on different computing resources impacts conflicts (interferences) on interconnect between computational resources and memory. This problem is increasingly important with the integration of various functionalities.

We propose to address this problem at different levels. First, we equipped Capella models with two kinds of reasoning capabilities. The first one is based on the worst case analytic evaluation of the interconnect interferences of a specific deployment (easy to compute but pessimistic). The second one is based on the (exhaustive) simulation and provides accurate interconnect interferences (more computationally intensive than the analytic methods but accurate). These reasoning capabilities help the designer considerably but he still has to explore several potential solutions by hand. To help him, we proposed a small DSL to express the exploration space from which the former reasoning can be performed automatically.

Amin Oueslati, Philippe Cuenot, Julien Deantoni, Christophe Moreno. *System Based Interference Analysis in Capella*. Licensed under Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). In Journal of Object Technology, vol. 18, no. 2, 2019, pages 14:1–21.
doi:10.5381/jot.2019.18.2.a14

We experimented with these techniques in the context of the ATIPPIC collaborative project, based on the modeling of simple but representative models in Capella.

Keywords Models, Operational Semantics, Interference Analysis

1 Introduction

The aerospace domain has a long tradition of dedicated hardware and software, tailored to space conditions and designed to be more resilient to SEU (Single Event Upset). Such design reduces the capabilities of embedded hardware and in many cases forced to disable some of its capabilities (for instance processor caches are disabled because they are very sensitive to SEU). Nowadays, better control of fault management techniques opens the door to Commercial Off-the-Shelf (COTS) hardware components for satellites. This makes it possible to define new computing architectures based on already existing hardware to enable the integration of various avionic features on the same hardware. These new architectures are essential for new low-cost satellites. However, beyond the study of fault tolerance mechanisms (important but not the focus of this paper), the computing power introduced by the use of COTS and the need to group various functionalities on the same hardware board makes it more difficult to control the deployment of the application on the architecture. For instance, it becomes more difficult to understand miscellaneous emerging behaviors such as, for example, the emergence of temporary bus congestion due to unexpected synchronizations between different tasks.

Model Driven Engineering [Sch06] (MDE) has been introduced to assist in system analysis at the early stages of the development process. MDE is increasingly being used and is nowadays a common practice in many software related disciplines [WHR14]. For example, Capella [BVNE] is a tool-based open source language that supports system architecture design (`http://polarsys.org/capella`). It was introduced by Thalès[1] and is used in many other companies. Capella is a great help in providing a consistent view of the system architecture, which can be reviewed, shared, etc.

Despite interesting features, Capella is not yet fully equipped to assist system designers with the understanding of emerging behaviors. This is mainly due to the generality of Capella, encompassing various disciplines, which forbids the definition of a full operational semantics from which simulation and behavioral analysis can be conducted. For instance, when defining new software and hardware architectures for satellite systems, it remains difficult to understand the impact of architectural choices at the early stages of the development process. This makes it impossible to explore different deployment and configuration solutions early on. In other words, despite the use of Capella models (*i.e.*, MDE), it is nowadays difficult to tame the adequacy between the application and the architecture at an early stage of the development process. This is a fortiori the case when new computing resources are required to integrate new functionalities on the same hardware platform. Note that this is not a pure classical scheduling problem (although it still needs to be resolved) but rather a *communication scheduling* problem since the interconnect between the different computational resources and the memory becomes a potential bottleneck that must be used wisely. To do so, it is important to adjust (1) the deployment of the different tasks of the system on the right computational resource and (2) to schedule their

[1]`https://www.thalesgroup.com/fr`

communication in such a way to avoid interference on the interconnect; *i.e.*, it avoids communications initiated by different tasks to use an interconnect at the same time, for instance by delaying the start of some communications at specific point in time. Of course, such decisions should be consistent with more traditional scheduling analysis, *i.e.*, with respect to periods and deadlines.

What we present in this paper is the use of Capella models allowing the exploration of different architectures, in terms of deployment and parameterization, regarding interference in interconnect. The models we used suitable for defining both hardware and software models, as required in the context of the ATIPPIC project (a collaborative industrial project). The main contribution is the development of a semantics adapted to two different but complementary approaches to compute the level of interference on interconnects of a systems. The first approach defines analytical method from which latency bounds can be obtained in an interconnect. While pessimistic, this gives a coarse grain idea at low cost,*i.e.*, without expensive computing, of possible interference on interconnect. The second approach defines an operational semantics for Capella (based on the GEMOC studio). Based on this semantics it is possible to run simulation, potentially exhaustive. These simulations allow computing interconnect usage and latencies in task communications due to interference. While it requires simulations, this method provides a fine grain understanding of interference in interconnects.

Finally, based on each of these methods, we provide another small contribution, a small Domain Specific Language (DSL) from which it is possible to specify the domain of the parameters we want to explore. Then, we automatically generate the different models for these domains, simulate them and provide a representation of the results to help the designer to choose the appropriate configuration.

The modeling concepts and technologies used for this study are described in section 2. Then a simple running example as tutorial for the proposed methods is presented in section 3, section 4 describes the implementation for analytic and operational solutions and debates on design exploration extension, with in section 5 the demonstration of the evaluation on the ATIPPIC avionic use case. Finally section 6 documents the state of the art of such approaches, before concluding in section 7.

2 Background

2.1 Modeling Technologies

Capella Capella is an open source Model Based System Engineering (MBSE) solution hosted at PolarSys working group of the Eclipse Foundation[2]. Capella provides formalisms and toolsets that implements the ARCADIA method developed by Thales [BVNE, Roq16]. The method defines a four-phase workflow: operational analysis and system analysis to identify operational and system level needs, logical and physical architectures to identify components that meet these needs. For each phase of the workflow, Capella provides a set of diagrams to support system description, such as functional data flow diagram to describe functions and their exchanges, functional chains diagram to identify functions necessary to realize a given requirement and scenarios to describe a sequence of messages exchanged over time.

In this paper we only focus on Physical Architecture description and more precisely on Physical Architecture Blank diagram (PAB). Indeed, PAB provides a suitable syntax for Hardware and Software co-modeling. However, as we target low level details of an

[2] `https://www.polarsys.org/`

hardware architecture, we still need to complete the model with micro-architecture specific information that is not covered by standalone Capella meta-models.

KitAlpha Kitalpha[3] is an set of eclipse plugins, based on Capella, that allow to extend Capella models with Domain Specific information. Also hosted in PolarSys repository, it enables customization of Capella syntax for a specific *viewpoint*. Developing a viewpoint allows to describe specific concerns on top of Capella's generic ones. For instance, this mechanism was used to define the specification of fault tolerant mechanism directly in Capella[4].

Gemoc Studio The GEMOC Studio is a set of eclipse plugins that provides generic components through Eclipse technologies for the development, integration, and use of heterogeneous executable modeling languages[5]. It embeds a set of metalanguages that allow to define the operational semantics of these languages. When a semantic definition is provided, it automatically generates an interpreter, an fully aware debugger and recently a compiler.

2.2 Modeling Interferences

The performance and deadline of an embedded system are mainly affected by the communication scheduling of the application and in particular by possible *interference* on access to shared hardware resources. We focus our approach on *data memory transaction* because it is the major factor on communication scheduling for an application.

What we call *Interference* is the result of a concurrent access to a bus. From a task point of view, Interference produces latency on bus communication interface, *i.e.*, memory transactions are delayed. In the following, we define bus interference as the duration during which more than one task attempts to access to the same bus. The duration is computed on an hyper period during which each task that uses the bus is executed at least once.

For example, let's consider two tasks on two different computing resources, of 1ms period. They both start their execution by reading data from memory for $30\mu s$. The first task $t1$ runs for $200\mu s$ and produces data to memory with a transaction that takes $20\mu s$. The second task $t2$ runs for $350\mu s$ and produces data to memory with a transaction that takes $40\mu s$. In this case, both $t1$ and $t2$ try to access to the memory bus at the same time as they start: there is an interference. Either $t1$ or $t2$ accesses the bus first and the other access is delayed, in this case for $30\mu s$. The other communication (writes) do not interfere since the execution time of the tasks are different. Therefore in this case the interference is equal to $30\mu s$.

Note that the interference rate can be calculated by dividing the interference by the total transfer time over an hyper period. For example in the previous example the interference rate $= \frac{30}{30+30+40+30} = \frac{30}{130} \approx 23\%$.

Such high level analysis is important because memory and interconnect components can be the bottleneck in the communication scheme of the application. So it is important to master the bus performance in a chip to detect and minimize the local interference.

[3]`https://www.polarsys.org/kitalpha/`
[4]`https://youtu.be/MXdZdCRDMH4`
[5]`http://gemoc.org/studio`

3 Running example

In this paper, we use the simplified example of Figure 1 to illustrate the two approaches we propose for interference analysis. Since it is not representative of real world software and hardware architecture, a concrete use is presented below. The example was done under the Capella tool, using the PAB diagram which was extended with a Kitalpha viewpoint (see section 4.1). The hardware architecture is composed of four Physical Components, two represents CPUs, the third is an interconnect and the last one is a memory. These *Physical Components* are connected by *Physical Links* which are considered as bus connections in our case of study. In addition, we allocate two *Physical Components Behavior* to represent the software tasks, one on each CPU. The data dependency between the two tasks is abstracted by the *Component Exchange Link* that directly connects the *Output Flow Port* of Task1 to the Input Flow Port of Task2. These output and input ports are then allocated on *Component executions Physical Ports* and the data dependency is explicitly translated into two transactions carried by two *physical Paths.* The first *Physical Path* in red consists of two *physical Links* and connects CPU1 to the memory passing through the interconnect. It represents the transaction path for writing Task1 data to memory. The second *Physical Path* in blue is also realized by two *physical Links* and connects CPU2 to the memory passing through the interconnect. It represents the transaction path for reading Task2 data from memory.

More generally, without considering all the Capella formalisms, Task1 and Task2 are both periodic (respectively 20ms and 30 ms) with offsets (respectively 0ms and 7ms. Task1 executes on CPU1 for a certain time interval (between BCET 9ms and WCET 12ms), then writes data (5 MBytes) to Memory using *cpu1_ to_ interconnect* (red link) and *interconnect_ to_ memory* (black link) buses. While Task2 is allocated on CPU2, reads data (5 MBytes) from Memory through *interconnect_ to_ memory* (black link) and *CPU2_ to_ interconnect* (blue link) buses and then runs for a certain time interval (BCET 5ms and WCET 7ms). The frequency and data width of the three bus are identical, 125MHz and 8Bytes, enabling a bandwidth of 1GBps. The execution scenario described is similar to the producer-consumer problem replacing the shared variable by a shared resource which is depicted in this example by the *interconnect_ to_ memory* bus (black link). In the following and for the sake of readability, we will avoid using Capella naming. We will only refer to diagram elements with the name of the concepts they represent.

4 Proposition

The proposal is to compute the *interference* and influence on *bus load* of the on-chip communication (buses) based on Capella models. These properties are mainly influenced by the timing of the component's uses of buses. To compute them it requires 1) to identify the concepts from Capella that are involved in communications and 2) to define the parameters of these concepts that affect bus communication schedules.

As the computation targets the early stages of the application design, we do not consider, in a first step, optimization features of components (e.g. interconnect memory buffers). Moreover local memory transactions such as data cache access and control from processor are not considered. We abstract such mechanisms and only consider communication with bus interfaces.

We focuses on specific parts of Capella model level; namely the Physical Archi-

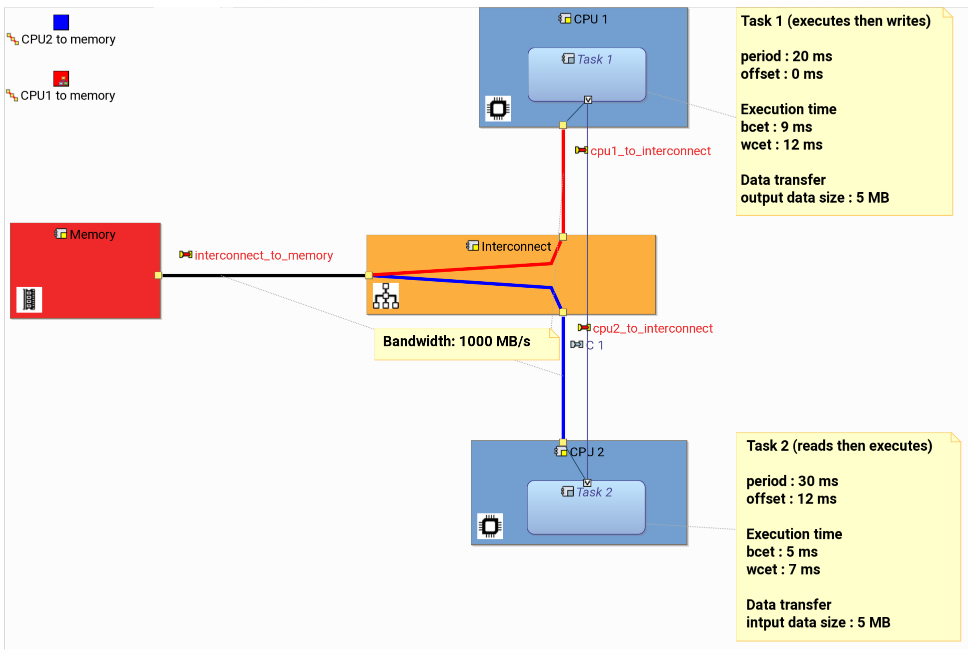


Figure 1 – Capella PAB running example viewpoint

tecture. Interesting elements of the Physical Architecture are represented in PAB diagrams, which represent allocation of functions into behavioral components mapped to hardware execution components. Applied to a SoC, this represents in one hand mapping of behaviors defined in software component to general purpose processors and in the other hand mapping of behaviors defined in hardware IP to hardware execution support (*e.g.*, Programmable Logic part). In other words, such diagram describes deployment of software architecture on an hardware architecture.

However, Capella's expressiveness is not rich enough to describe all domain specific properties required by our analysis. Consequently, in order to define their execution semantics, we identified in the following, a set of concepts on which properties for analysis are defined. Note that these concepts are mainly inspired by the UML Profile for MARTE [OMG09] especially the Hardware Resource Modeling (HRM) and Software Resource Modeling (SRM) packages and are mapped onto Capella (see section 4.1):

- **Execution Components (tasks):** Components representing software or hardware implementation of tasks. They are the initiators of data traffic. These components follows a read-compute-write semantics, meaning that all readings from memory are done before the computation and all writings to memory are done after the computation.
- **Computation resources:** General purpose processors and programmable logic units, on which tasks are allocated. They provide computational capabilities and communication interfaces for tasks.
- **Communication resources:** Components responsible of routing and transferring data, like for instance Buses, Interconnects and IO interfaces.

- **Memory resources:** Component representing data storage. They are sources or destinations of some *transactions*, used as data storage and data exchange zone for tasks.
- **Sensors/actuators:** Components respectively representing first data providers and last data consumers.

Based on these concepts, we select for each of these components a set of properties and parameters that are relevant for bus performance analysis. Conflicts on the interconnect occurs when two or more data memory transactions use the same bus at the same time. Thus time properties have a significant impact on interferences and associated latencies. We consider timing properties related to concepts previously identified, which have an impact on the communication scheduling. We propose the main following properties, organized according to the concepts defined previously:

- *Component execution (tasks):*
  - Trigger: This enumeration defines whether task is triggered periodically or triggered upon receipt of a specific event on a task port.
  - Period: When the task trigger is periodic, its period specifies the task execution rate.
  - Offset: When the task trigger is periodic, the offset represents the date from which the task is executed.
  - ExecutionTime: Defines the duration for which the task is executed. It is defined by an interval specifying the best and worst execution time (BCET, WCET)
  - DataSize: Defined by inputDataSize and/or outputDataSize, which respectively defines the size of data read before the execution and the size of data written after the execution.
  - DataPath: Union of an input data path and an output data path, specifying the succession of buses (the route) used respectively to read and write data from and to a memory.
- *Communication resources (buses):*
  - InterfaceSize: Defines the size of packets that can be sent by the communication resource.
  - Frequency: Represents the speed at which packets are handled by the communication resource.
- *Sensor:* Holds same properties as component execution but only those related to data production.
- *Actuator:* Holds same properties as component execution but only those related to data production.

Finally concepts identified previously but not carrying properties (*e.g.*, Memory resources) are used to ensure structural correctness of models. For instance we can check that a data path is started or ended by a memory resource, thus avoiding two other types of component from communicating without storing data in memory. Note that such path can be derived from the existing notion of Physical Path in Capella. Same kind of structural correctness can be verified on sensors and actuators.

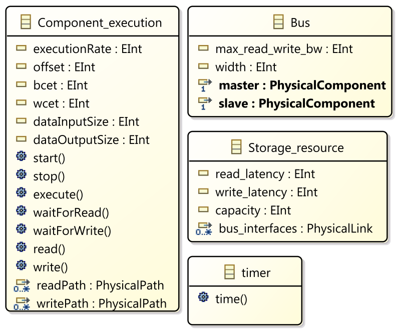


Figure 2 – Kitalpha extension Class Diagram

4.1 Extending the Capella model

In Capella, for the description of components and properties set, we specialize the abstraction level of PAB objects to include missing information. To do so, Kitalpha tool is used to generate viewpoint extension.

On the running example of Figure 3, we distinguish three different kinds of Physical Components, computation resources for CPU 1 and 2 (in blue), a bus controller for interconnect (in orange) and a memory resources (in red). Physical Links mapped on buses Physical Components Behavior mapped on component execution are in also depicted in section 3.

In addition to properties, Kitalpha also provides the possibility to extend the model by defining operations describing actions associated and realized on each element. These operations are needed for simulation of operational scenario. Figure 2 shows the tab view generated by Kitalpha for setting components properties and calling behavioral methods.

In our study, we only implemented a subset of hardware and software components and properties enough to cover our analysis. We are aware that for the sake of compatibility and standardization, a better solution would be to rely on an existing solution (*e.g.*, by using the MARTE profile as implemented in Time4Sys [6]). This may be done in the future but it requires efforts to adapt the behavioral semantic layer to the selected profile.

4.2 Analytic Solution

The first approach we propose is based on analytic evaluation of effect of memory transaction on architecture to estimate interference (latency) on software elements (task) and hardware elements (bus), and to evaluate bus occupation. Only static context is considered meaning following assumptions on memory transactions: 1) all transactions are atomic 2) the interconnect arbitration is not considered 3) all transactions crossing a bus port interface are concurrent. As a consequence, performance properties like delays and interferences are computed under worst case hypothesis. While possibly pessimistic, calculated values can be used as bound to have a first idea of the design performances all along the design activity.

The computation of the various performance properties are defined as follows. We first provide intermediate definitions used as helpers in the computation:

[6] `https://www.polarsys.org/time4sys/`

- let T be the set of all tasks in the system.
- let T_p be the set of tasks that make use of a specific *port*. It is defined as follows:

$$T_p = \{\, t \in T \mid t.dataPath \rightarrow contains(p) \,\}$$

- let $Bandwidth_{bus}$ (MB/s) be defined as follows

$$Bandwidth_{bus} = Frequency_{bus} \times InterfaceSize_{bus}$$

Let now define how the different performance properties are computed:

1. $transferTime_{task}$ (μs) := (the time spent to carry data in a period.)

$$\frac{(task.outDataSize + task.inDataSize)}{min(task.port.allocation.Bandwidth)}$$

2. $maxDelayedTime_{task}$ (μs):= (the maximum time during which a task can be blocked by other tasks using the same port in a period.)

$$\text{let } allTp_{task} = \{\, T_p \mid p \in task.ports.allocation \,\} \text{ in}$$
$$\text{let } hyperPeriod_{bus} = LCM(t.period \forall t \in allTp_{bus}) \text{ in}$$
$$\max_{ct \in allTp_{task}} \Big(\big(\sum_{t \in ct} \frac{hyperPeriod_{bus}}{t.period} \times transferTime_t \big) - transfertTime_{task} \Big)$$

3. $maxInterference_{bus}$ (μs):= (the maximum time during which the bus can have data blocked on its interface computed in an hyper period.)

$$\text{let } allTp_{bus} = \{\, T_p \mid p \in bus.ports.allocation \,\} \text{ in}$$
$$\max_{t \in allTp_{bus}} \Big(maxDelayedTime_t \Big)$$

4. $load_{bus}$ (%):= (occupation of the bus computed in an hyper period.)

$$\text{let } allTp_{bus} = \{\, T_p \mid p \in bus.ports.allocation \,\} \text{ in}$$
$$\text{let } hyperPeriod_{bus} = LCM(t.period \forall t \in allTp_{bus}) \text{ in}$$
$$\frac{\sum_{t \in allTp_{task}} TransferTime_t}{hyperPeriod_{bus}}$$

The calculation have been performed with parameters definition in Kitapha and java code for computing the formula. For more information on Kitalpha viewpoint see section 4.1.

The analytic method applied on the running example of Figure 1 allows $maxInterference_{bus}$ and $load_{bus}$ to be calculated when accessing to the memory component. First, the $Bandwidth_{bus}$ is equal to 1000 MB/s for all the buses, (*i.e.*, $Frequency_{bus}$ (125 MHz) $\times$ $InterfaceSize_{bus}$ (64 bits = 8 Bytes) = 1000 MB/s. The $TtransferTime_{task1}$ computed on CPU1 gives task1.outDataSize (5MB) / min(task.dataPath.bandwidth) (1000) = 5/1000 = 5ms, and $transferTime_{task2}$ = 5ms (5/1000).

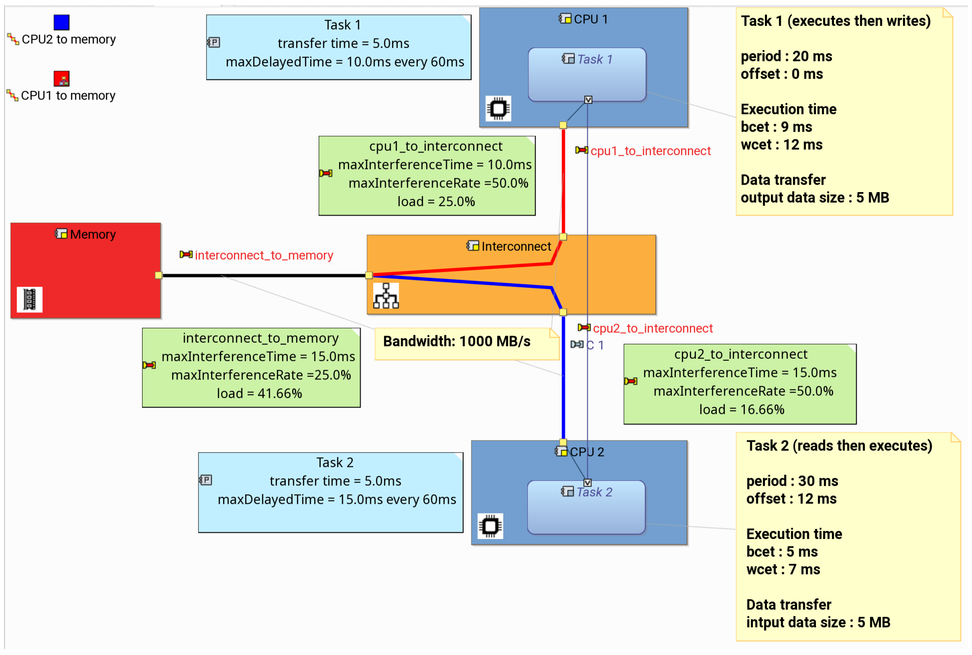


Figure 3 – Running Example from Figure 1 With the Analytic Computation Layer

The $maxDelayTime_{task1}$ is max($hyperPeriod_{bus}$ (LCM(20,30)=60) / t.period (30) ×transferTime$_t$ (5, 5) - $transferTime_{task1}$ (5)) = max (60/30×5) = 10ms and by analogy $maxDelayTime_{task2}$ is equal to 15ms. Applying the formulas:
$maxInterference_{Interconnect\_to\_Memory} = (maxDelayTime_t(10, 15)) = 15ms\ (rate = 15/60 = 25\%)$
$maxInterference_{cpu1\_to\_interconnect} = 10ms\ (rate = 10/20 = 50\%)$
$maxInterference_{cpu2\_to\_interconnect} = 15ms\ (rate = 15/30 = 50\%)$
$load_{Interconnect\_to\_Memory} = transferTime_t(5 \times 3 + 5 \times 2)/hyperPeriod_{bus}(60) = 25/60 = 41.66\%$
$load_{cpu1\_to\_interconnect} = 15/60 = 25\%$ and $load_{cpu2\_to\_interconnect} = 10/60 = 16,66\%$

We have extended Capella views with a layer that computes these values to help the designer to understand the impact of its design choices on the performances. Figure 3 shows a view of the result based on the running example.

4.3 Operational Solutions

The second approach we propose, is based on model simulation. As we want to reason at high-level of abstraction, we use the Capella model as a reference for our simulation, which allows us to evaluate interferences from the model parameters. So we used GEMOC studio facilities to define an operational semantics of the Capella PAB diagram for interference estimation.

In our model, communications are initiated by tasks. In most cases, a task reads data from memory, executes itself and writes back results to memory. Some tasks are scheduled periodically and others are triggered as soon as their input data are available. We do not consider a fixed execution time for tasks, instead execution time is randomized between its best and worst case. However when on a read or write access the bus may be busy by another on-going transaction. Then the task should wait for the bus until the current transaction is complete which generates latency. Thus, a communication bus can only process one transaction at the same time, it is considered busy until the end of the transaction.

4.3.1 Encoding the behavior of the system under Gemoc

In the Kitalpha viewpoint, we first need to define the set of operations which describe the system behaviour. In the case of component execution (task), we define the following operations: *start(), stop(), execute(), read(), write(), waitForRead()* and *waitForWrite()* as shown in figure 2. Once we defined these operations, we define the actions to be performed in each operation and the dynamic information required to monitor the evolution of the system during execution. To illustrate this, let's consider the case a task waiting for a bus: the *wait()* operation is thus executed increasing the bus latency counter and updating the value of the bandwidth. In Gemoc Studio, dynamic information are called Runtime Data (RTD) and execution function, defined for the Domain Specific Actions (DSA), are implemented in Kermeta[7]. A DSA implements the execution semantic for operation defined from system data.

The second step is the implementation of the control flow semantic. We first define the Domain Specific Events (DSE) which trigger the execution functions (from DSA). We may have several instances of a concept (e.g. in the running example of figure 1 there is two instances of task). For each instance Gemoc generates a Model Specific Event (MSE). For example, in the context of task, we define a *start, stop, execute, read, write, waitForRead* and *waitForWrite* DSE. Applying this to the running example, in which we have two task instances, Gemoc will generate an instance of the corresponding DSE for each task as following: *MSE_ Task1_ start, MSE_ Task2_ start, MSE_ Task1_ stop, MSE_ Task2_ stop*...etc. A MSE is an ordered set of event occurrences that will execute the associated DSE function instances. To ensure a correct execution of the system, MSE occurrences should trigger the execution functions in a specific order (e.g. the start of a task occurs before the execute). This order is obtained by setting constraints between DSE events, using specific *invariants* defined in CCSL[And09] (Clock Constraint Specification Language), and in MoCCML (Model of Conccurrency Modeling Language) extension. By reasoning on temporal properties of the DSE, these two languages allow us to define the order in which MSE events occur. MoCCML expressions can be implement by automata for more complex scenario.

The built model semantic includes six types of tasks, each type having its own execution semantic. Four of them are scheduled periodically and the two others are data triggered. For the task execution semantics, the event schedule is defined by the physical time requirement (e.g execution time included between a BCET and WCET given in μ). It is necessary to take physical time into consideration when defining the model semantic. The different task behaviors are described as following (a wait can preceded read and write when bus is busy):

1. Starts, reads data from bus, executes, writes data on bus and stops, scheduled periodically.
2. Starts, reads data from bus, executes and stops, scheduled periodically.

[7]http://diverse-project.github.io/k3/

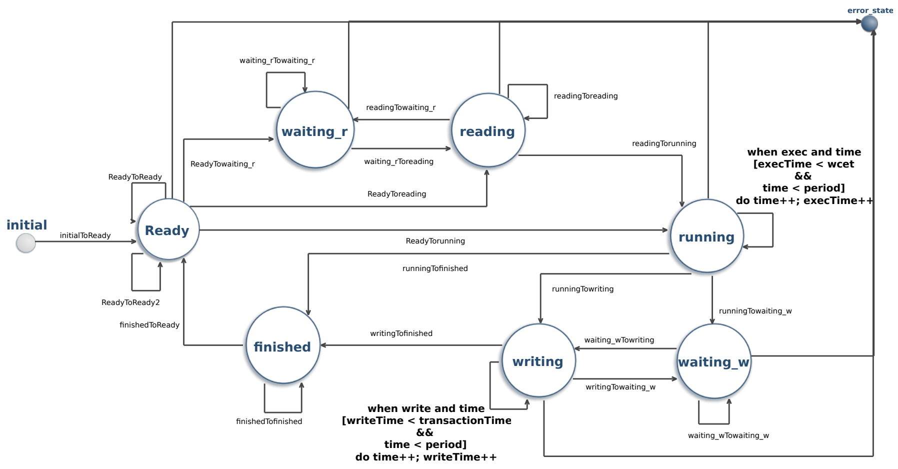


Figure 4 – Periodic task semantic in MoCCML

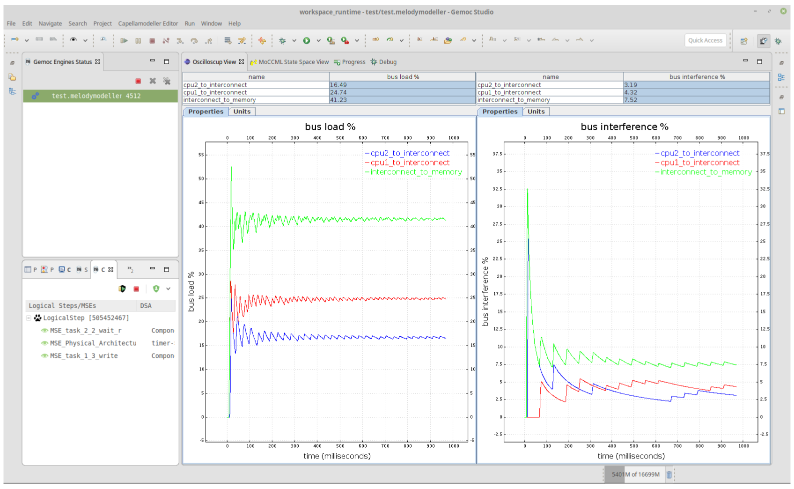
3. Starts, executes, writes data to bus an stops, scheduled periodically.

4. Starts, executes and stops, scheduled periodically.

5. Starts, executes, writes data to bus and stops, scheduled when data dependency is satisfied.

6. Starts, executes and stops, scheduled when data dependency is satisfied.

We implement the above semantics using two MoCCML automata, one for periodic tasks and another for data triggered tasks. Each transition in the automata triggers a *time* event. The physical time is build according time event scheduling build with resolution from our system requirement (1μ). Figure 4 shows the automata describing the periodic tasks behaviors. A periodic task starts in the *Ready State* and waits until its offset decount reach 0. Depending on the bus state (taken or idle), a task can read data from bus, wait for the bus or execute. In the MoCCML automata, this is managed by taking one of the three transitions leading to *waiting_ r state*, *reading state* or *running state*. Choosing one transition depends on its associated conditions and events of the DSE which is triggered. For instance, when transitioning from *ready* to *reading*, the conditions are task of type 1 or 2 and *dataInputSize* > 0, and the associated event as the *read MSE* of the task. However, we also to constrain all the *read* and *write* occurrences on the same bus, as it is impossible to have more then one task communicating on the same bus. A CCSL constraints is used to exclude all the read and write combinations allocated to the same bus. In the context of the running example, we exclude all occurrences of Task1_read from triggering simultaneously with *Task2_ write*, forcing one of the two tasks to trigger its *wait* event. Once a task is in the *reading state*, task triggers *read* event and stays in reading until the bus transfer time is completed. When the task read ends, it enters in the *running state* by triggering the execution event of the transition. As the execution time is included in BCET and WCET interval, the *execution* timing event is randomize leading to different execution time from one period to another. The *writing state*, is similar to the *reading state* with the possibility of going into *waiting state* if the bus is busy. Finally,



name	bus load %
cpu2_to_interconnect	16.49
cpu1_to_interconnect	24.74
interconnect_to_memory	41.23

name	bus interference %
cpu2_to_interconnect	3.19
cpu1_to_interconnect	4.32
interconnect_to_memory	7.52

Figure 5 – Results of simulation under Gemoc environment

task enters to the *finished state* by triggering the *stop* event and continues to consume time until it reaches the next period's schedule. In this MoCCML automata, each transition event is linked to a MSE event. Triggering an event causes the execution of the associated execution function in the DSA, updating the runtime data for changing the system state.

4.3.2 Simulation and results analysis

The operational method applied to the running example of Figure 1 evaluates $InterferenceRate_{bus}$ and $Load_{bus}$ by simulating the model. We have developed different scenarios in which we vary the value of Task2 offset. In some scenarios, tasks are no more schedulable due to interference effects induced by the task execution time which varying randomly between BCET and WCET. For instance, if the $InterferenceRate_{CPU1\_to\_Interconnect}$ is greater than 15%, and if Task1 executes for 12ms, then Task1 cannot end before its 20ms deadline. To validate the system requirement, we generate different simulation scenarios based on different hypothesis on system parameters (task scheduling, data decomposition or aggregation) which allows to estimate $InterferenceRate_{bus}$ and $Load_{bus}$ value. Figure 5 shows the results of simulation of the running example under Gemoc Studio for a configuration where the offset of Task2 is fixed to 9ms.

If we compare the results between the operational and the analytic solution on the running example, we can conclude that 1) the non schedulable scenario cannot be detected by analytic solution, 2) the schedulable scenario of operational solution is always bounded by value calculated in analytic scenario.

4.4 Design Space Exploration

In previous sections we show how we equipped Capella with analytic and simulation based reasoning capabilities. Based on them, we can retrieve information helping the designer to evaluate the quality of system design candidate. However, at early stage of the development process, some characteristics of the system may not be totally known, for instance the exact kind of hardware bus and its performances. Also, the

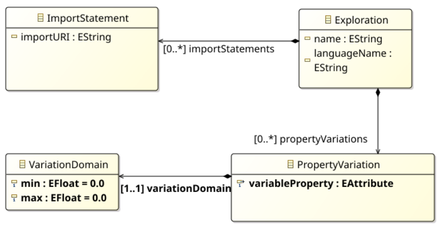


Figure 6 – Abstract syntax of the little DSL for Design Space Exploration

algorithm used by the task and their scheduling properties may change. If we consider a camera-based system, different compression algorithm may be used. Some of them take more time to compute but require less data to transfer (better compression) while others are faster but produce more volume of data. Exploring all these possibilities manually may be painful for a designer. Consequently we proposed a small DSL dedicated to defining the potential solutions to be explored. Then, from such a description we automatically generate the models with the appropriate characteristics that we can simulate (possibly in parallel) to obtain a whole sets of simulation results. These results can then be explored in different manners. In our experiments we used Jupyter[8], an in the web notebook, to easily explore the results of the simulations.

4.5 DSL for Domain Space Exploration

Our DSL (whose abstract syntax is represented Figure 6 is adaptable to any EMF model and proposes to represent the range of variation of different model attributes. This DSL remains very simple for our current use but is subject to several evolutions.

An *Exploration* is importing a model by using an *ImportStatement*. This allows to access to all elements of the model to be explored. An important point is that an exploration defines the *languageName* with which the exploration must be conducted. This is important since it indicates which semantics should be apply to drive the simulation. Then, for each element in the domain to be explored, the user creates a *PropertyVariation* with reference to an *Attribute* from the initial model into a reference named *variableProperty*. Finally, a *VariationDomain* on this property is defined as a Float interval. This clearly means that only Float compatible attributes can vary during our exploration. We have several extensions of this DSL under study (supporting different data types but also topology variation, for instance to represent different allocations of the tasks on the hardware); but this simple DSL was expressive enough in our initial ATIPPIC use case.

To make it usable we defined a textual concrete syntax using Xtext[9]. An example of the use of this syntax is given in Figure 7. In the tooling, we provided efforts to ensure that completion is enabled for both the choice of the language and the navigation to the model attributes, helping users to make a correct exploration model.

4.6 Exploitation of an Exploration Model

Once a designer defined an exploration model, it can be used to export a set of executable artifacts, each generated according to different configuration of the model

[8]`https://jupyter.org/try`
[9]`http://eclipse.org/xtext`

```
Exploration simpleExplorationOfRunningExample

import "platform:/resource/com.irt.atippic.simpleExample/runningExample.melodymodeller";
with "com.irt.atippic.atippic_contention_model.xdsml.atippic_contention_model"

Variation of "task 2.offset" in [10.0 ; 15.0]
Variation of "task 1.bcet" in [5.0 ; 7.0]
Variation of "task 1.dataOutputSize" in [4.0 ; 8.0]
```

Figure 7 – a Simple example of use for our DSL

depending on the property variations defined in the exploration model. More precisely, based on the Cartesian product of the different values covering the exploration domain of each property variation, a model is generated and then compiled into a java Class allowing its *monitored* execution. By monitored execution, we mean that it is possible, by changing arguments of the main function, to log values of interest during the simulation in a csv format. With the set of executable artifacts, a script to launch the different executions is generated. During the execution of each model, a csv file and a corresponding gnuplot script is generated so that it is possible to have a first view of the results.

Additionally, to make easier the exploration of the results, we used jupyter lab, which provides, among other things, an easy way to generate a dedicated interface to browse the resulting csv files according to the parameters. A screen shot of the resulting view is provided in the companion webpage.

5 Case study

The avionic use case developped in the ATIPPIC project targets the market for low cost earth observation or communication mini satellites cruising at Low Earth Orbit (LEO) build upon COTS (non Rad-Hard components). The avionic supports standard featuresto perform its control and maintenance, and embeds a payload. The satellite control is operated with a star tracker sensor to determine the orientation (or attitude) of the spacecraft with respect to the stars, a GNSS compatible with GPS or Gallileo band to acquire its positioning, standard RF communication means for Telemetry/Telecommand (TM/TC) actions, and internal communication interfaces as CAN bus or SPaceWire (SPW) used to acquire sensors signals and to actuate the satellite propulsion, or the energy management with solar panel orientation. The SPW or CAN communication allows the integration of an application payload. For the sake of the study, the avionic use case is completed with an earth observation payload application. This is a typical use case supported by as the ATIPPIC On Board Computer(OBC) which offers extensive functionalities for payload integration such as storage capacity though mass memory implementation and fast RF communication links to download image to ground through Telemetry Image interface (TMI). The On Board Computer (OBC) architecture is based on redundant architecture, not scope of the analysis and so not detailed here, embedding a SoC as main computing processor. The objective of this study is to evaluate in the early stage of development of the system architecture, how to balance control and application between the Programmable Logical (PL) part and the CPU of a SoC (to reduce implementation constraints on the PL area), by analyzing the required timing requirement of the communication scheme to detect possible overloads of the internal bus communication.

The analysis is performed on a sub-part of the overall system architecture comprising an on-board avionic to control and acquisition of the signal of three optical head via

SPW interfaces, to format the raw data performing star tracker resolution and to reconfigure its heads if necessary. The payload includes the acquisition of raw images through SPW interface, data formatting and compression for storage in an external flash mass memory (controlled by an integrated memory controller implemented in the PL area), and transmission to the TMI interface connected to an external RF TMI-X transmitter.

The OBC electronic is build with a SoC from the Xilinx Zynq7000 family offering PL capabilities. The SoC is decomposed with a Processing System area (PS) including a dual CortexA9 processor in the Application Processor part, a central interconnect connected to a set of I/O devices and a memory interconnect to allow access to external DDR memory device. Note that DDR access is also possible directly from L2 cache controller of the dual CortexA9 or from the central interconnect. The PL provides a user configurable area to allow the integration of the required hardware IP components. The PL is connected to the PS via the central interconnect with AXI General Purposes ports (GPx) and via the memory interconnect with AXI High Performance ports (HPx) to access to DDR.

The use case architecture, see Figure 8, manages access to DDR from different source: from the PS by the *CorteXA9* cores via the *SCU*, or from PL area by the *swp_ IP* addressing the *AXI_ HP0* port of the *DDR_ Interconnect*, or by the *spw_ payload_ camera_ IP* addressing the *AXI_ HP2* port of the *DDR_ Interconnect*. For the PL part, the memory transaction flows have been segregated on separated slave port of the *DDR_ Interconnect*, visible by Grey Physical Link in the Figure.

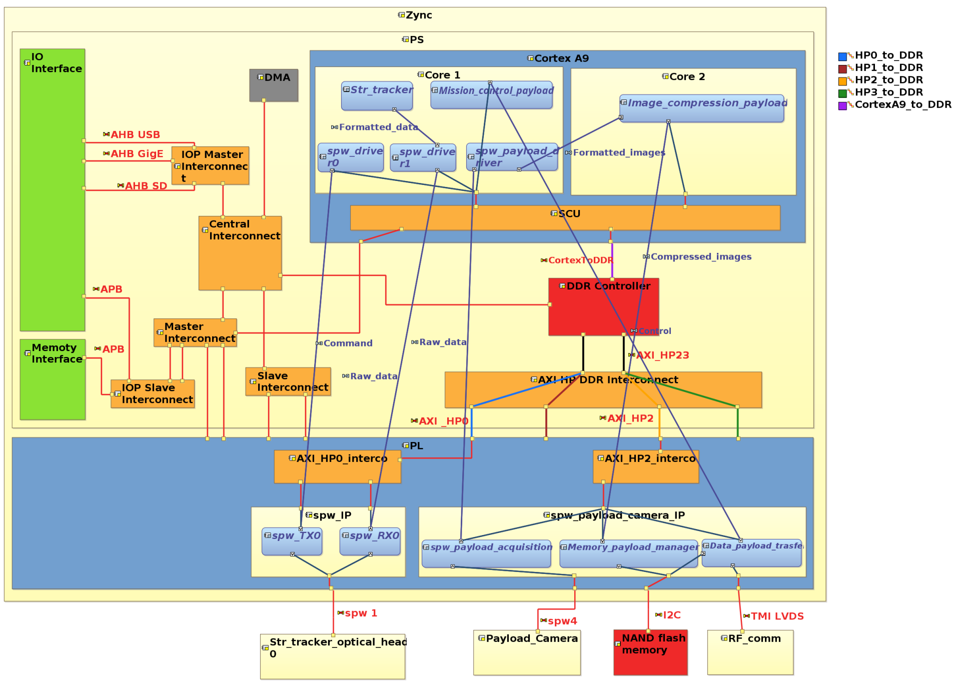


Figure 8 – Use Case System Architecture

In this architecture interference may occurs locally on following component :

- ***AXI_ HP2_ Interco*** for the image control since the transfer of acquired raw images (1,2 Mb acquired every 0.01s) to the DDR realized by *spw_ payload_ acquisition* can be concurrent with the Flash writing of the compressed image (factor 2) from the DDR to the Nand Flash controlled by Memory_payload_manager.

- ***SCU*** and inside the ***DDR Controller*** as image compression *Image_ compression_payload* and image formatting *spw_payload_ driver* can interfers with access to image data because they are assigned to different cores of the CortexA9, and executed asynchronously as the compression process is slowest then the driver's image formatting. This design choice is motivated by future extensions of image processing functions, and by the reservation for new functions implemented in the PL area but not described in this use case.

The timing requirement of the architecture must be evaluated and we provide the means to assess them early in the design by identifying the software latency on tasks $DelayTime_{task}$, the AXI bus hardware latency and load respectively $Interference_{bus}$ and $Load_{bus}$. This allows to challenge our design, offering the advantage of relaxing PL occupation size, and giving the means to compare alternative solutions. Note that alternative solutions may also vary depending on the configured scheduling of software tasks, or on the reassignment of software tasks into the PS.

In the two graphs of Figure 9, we compare bus average interference value (total interference duration on each bus divided by the number of transaction achieved by the bus) generated for two different approaches of image compression. In the first strategy, we consider a 2 by 2 image compression strategy. The second strategy compresses the images by group of 4. The first strategy executes more often than the second one and generates smaller transactions. While the execution rate of the second is lower and exchanged data size is larger. The results show that the first approach generates interference only on *CortexToDDR* bus (linking the DDR and the SCU of Cortex A9), less than the 4 images compression strategy which besides generating interferences on the *CortexToDDR* also generates interferences on the 2 other bus (the 2 buses linking PL to DDR through the interconnect). More information and simulation results based on the use case can be found on the companion webpage:`https://project.inria.fr/interferenceanalysis/`

6 Related Works

There exist many tools for network simulation (*e.g.*, NS3 [Car10] or OMNET++ [VH08]). However, these tools are used for accurate simulation of network protocol and usually does not focus on the node, viewed only as traffic generators. Consequently in the following we consider only platform close to our domain (embedded systems). Also, we did not tried to make some guided design space exploration like in [BZS18] or equivalent but these methods are compatible with ours, their fitness function requiring a simulation or the evaluation of properties by using the analytic method. Consequently we do not compare to them.

Platform Architect[10] distributed by Synopys provides an industrial solution to perform SoC architecture analysis and optimization for performance and power. It is based on SystemC TLM libraries with accurate modeling for interconnect, memory controller and virtual processor (accurate memory transaction definition and resource consumption). It allows through simulation and trace analysis to evaluate performance of a multicore or SoC architecture. This tool is used by SoC designer to optimize their

[10]`https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html`

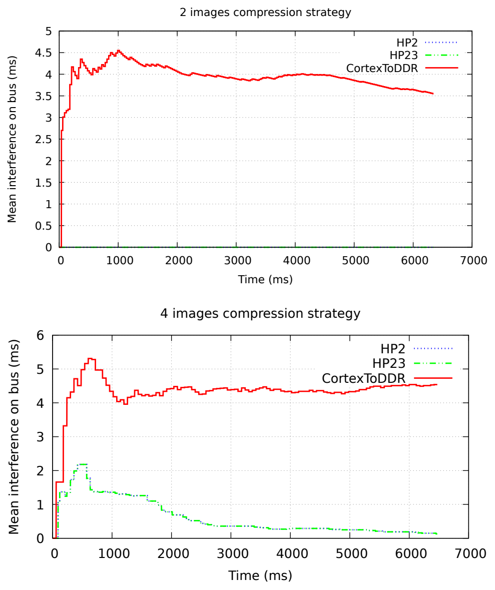


Figure 9 – Results of simulation for 2 compression strategies

design. However, compared to our approach, analysis requires strong hardware skills. It seems not adapted to the system design level and does not interface with the MBSE system design methods and tools, to our knowledge.

TTool\Diplodocus[AB12, AMAB+06, GA16][11] is a modelling tool based on UML/SysML, developed by Telecom ParisTech. One of its extension is a tool for SoC partitioning by finding the best candidate software and hardware architecture for executing a set of functions. It encodes a semantic for software execution derived from task graphs and uses concept for hardware definition (CPU, memory, interconnect, router..), whose parameters can configured, allowing exploration. The simulation is done by translation of the model element to (predefined) SystemC blocks. Compared to our approach it requires to have the predefined SystemC block and their execution environment which is realized outside of the modeling tool. Consequently it does not take benefit of model level simulation, debugging and exploration.

7 Conclusion

In this paper we have presented a use of model driven engineering that enables reasoning on bus interferences of an aerospace integrated architecture. We have identified the required information for such analysis. The analysis is based either on an analytic method, which provides instantaneous but pessimistic results or on a simulation based method, which provides more accurate results but require a simulation step. To help the designer in choosing the best parameters for its architecture we provided a small DSL based on which he can define exploration space. The exploration space is used to generate different instance of models from which reasoning can be conducted. Results are then presented in a small dedicated interface in a notebook.

[11] `https://ttool.telecom-paristech.fr/diplodocus.html`

Many future works are envisioned. Here is a few of them: the use of new design properties to allow more accurate results (caches, bus arbitration, DMA, etc); the use of more powerful analysis based on exhaustive simulations to guarantee temporal properties; the introduction of stochastic information (about execution time or data size) to provide stochastic results; the integration of the exploration DSL and/or results in dedicated exploration tools based, for instance, on parallel coordinates chart.

References

[AB12] Ludovic Apvrille and Alexandre Becoulet. Prototyping an Embedded Automotive System from its UML/SysML Models. In *4th European Congress on Embedded Real Time Software and Systems (ERTS 2012)*, TOULOUSE, France, January 2012.

[AMAB$^+$06] Ludovic Apvrille, Waseem Muhammad, Rab´ea Ameur-Boulifa, Sophie Coudert, and Renault Pacalet. A UML-based Environment for System Design Space Exploration. 2006.

[And09] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009. URL: `https://hal.inria.fr/inria-00384077`.

[BVNE] Stéphane Bonnet, Jean-Luc Voirin, Véronique Normand, and Daniel Exertier. Implementing the mbse cultural change: Organization, coaching and lessons learned. *INCOSE International Symposium*, 25(1):508–523. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2015.00078.x`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.2015.00078.x`, `doi:10.1002/j.2334-5837.2015.00078.x`.

[BZS18] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. Mdeoptimiser: a search based model engineering tool. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 12–16. ACM, 2018.

[Car10] Gustavo Carneiro. Ns-3: Network simulator 3. In *UTM Lab Meeting April*, volume 20, pages 4–5, 2010.

[GA16] Daniela Genius and Ludovic Apvrille. Virtual Yet Precise Prototyping: An Automotive Case Study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, pages 691–700, TOULOUSE, France, January 2016. URL: `https://hal.archives-ouvertes.fr/hal-01291888`.

[OMG09] OMG. *UML Profile for MARTE, v1.* Object Management Group, Nov. 2009. formal/2009-11-02.

[Roq16] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, January 2016. URL: `https://hal.archives-ouvertes.fr/hal-01258014`.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and . . . , 2008.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

About the authors

Amin Oueslati is an enginner at the IRT Saint Exupery, Sophia Antipolis, France. He was graduated in 2013 from Polytech Nice Sophia (Master degree in computer science). Former research engineer at INRIA Kairos Team. He moved to IRT Saint Exupery in 2018 and joined the ATIPPIC Project. He is currently working on research topics involving MBSE methods.

Contact him at `amin.oueslati@irt-saintexupery.fr`.

Philippe Cuenot is Research Engineer at IRT Saint Exupéry, Toulouse, France (seconded from Continental Automotive France). He was graduated in 1989 from ISTG Ploytech Grenoble French University (engineering diploma in Industrial Computer Science and Instrumentation). He joined Continental Automotive France (formally Siemens Automotive France) for development of engine management system real time software. In 2005 he moved to the Electronic Advanced Development team as innovation project leader on system and software methods. Since 2014, he is in delegation to the IRT Saint Exupery in System engineering department. Contact him at `philippe.cuenot@irt-saintexupery.fr` .

Julien Deantoni is an associate professor in computer sciences at the University Cote d'Azur. After studies in electronics and micro informatics, he obtained a PhD focused on the modeling and analysis of control systems, and had a post doc position at INRIA in France. He is currently a member of the I3S/Inria Kairos team (`https://team.inria.fr/kairos`). His research focuses on the join use of Model Driven Engineering and Formal Methods for System Engineering. More information at `http://www.i3s.unice.fr/~deantoni/`.

Christophe Moreno is Project manager IRT Saint Exupéry, Toulouse, France (seconded from Thales Alinea Space). He was graduated in "Computers science for industrial applications" from ENSEA in 1987. After a period as consultant in SW development in different domain as transport, production, military where he developed a real time operating system for Tigre Helicopter, he joined Thales Alenia Space in 1994 . He developed Ostrales RTOS as reference RTOS for TAS satellite since Proteus-Jason satellite (2000). He occupied the role of SW project manager on the first Proteus Observation

Satellite and the first Telecom Spacebus 4000 satellite before to take skill management function in TAS organisation. He joined IRT Saint Exupery in 2017. Contact him at `christophe.moreno@irt-saintexupery.fr` .

Acknowledgments The authors thank all people and partners involved in the ATIPPIC project managed in the IRT Saint-Exupéry Sophia, and in the GLOSE project. This work performed in the ATIPPIC project context and is supported by the French Research Agency (ANR) and by the industrial partners of IRT Saint-Exupéry Scientific Cooperation Foundation (FCS).