Microprocessors and Microsystems 78 (2020) 103244

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Mega-modeling of complex, distributed, heterogeneous CPS systems

Eugenio Villar [a,*], Javier Merino [a], Hector Posadas [a], Rafik Henia [b], Laurent Rioux [b]

[a] University of Cantabria, Santander, Spain
[b] Thales TRT, Palaiseau, France

A B S T R A C T

Model-Driven Design (MDD) has proven to be a powerful technology to address the development of increasingly complex embedded systems. Beyond complexity itself, challenges come from the need to deal with parallelism and heterogeneity. System design must target different execution platforms with different OSs and HW resources, even bare-metal, support local and distributed systems, and integrate on top of these heterogeneous platforms multiple functional component coming from different sources (developed from scratch, legacy code and third-party code), with different behaviors operating under different models of computation and communication. Additionally, system optimization to improve performance, power consumption, cost, etc. requires analyzing huge lists of possible design solutions. Addressing these challenges require flexible design technologies able to support from a single-source model its architectural mapping to different computing resources, of different kind and in different platforms. Traditional MDD methods and tools typically rely on fixed elements, which makes difficult their integration under this variability. For example, it is unlikely to integrate in the same system legacy code with a third-party component. Usually some re-coding is required to enable such interconnection. This paper proposes a UML/MARTE system modeling methodology able to address the challenges mentioned above by improving flexibility and scalability. This approach is illustrated and demonstrated on a flight management system. The model is flexible enough to be adapted to different architectural solutions with a minimal effort by changing its underlying Model of Computation and Communication (MoCC). Being completely platform independent, from the same model it is possible to explore various solutions on different execution platforms.

1. Introduction

Embedded Systems (ESs) are the fundamental constituents of the Internet of Things (IoT), the new paradigm leading the development of electronics in the medium-term. Based on them, it is possible to conceive the design of new electronic systems providing the computing and communication resources required by new applications [1]. Nevertheless, the design of these new applications has to cope with their increasing complexity as the main problem to be overcome. Design complexity comes from three main reasons:

- Increasing complexity of the integrated circuits on which the embedded systems can be implemented, with a higher number of processing cores, co-processors, peripherals, sensors, etc.
- Increasing heterogeneity of the computing platforms with a larger variety of computing resources of different kind, such as CPUs of various types, either with the same instruction-set (i.e. big-little) or different (i.e. ARM-RISCV), GPUs, DSPs, application-specific HW, etc.
- Increasing number of computing nodes in the execution platform as, in order to be able to provide the required services, the application has to be distributed over a network of nodes of different kind, from simples sensing motes and embedded systems to computing resources in the fog and the cloud. In some extreme cases, the system being design has to be conceived as a collection of systems; a System-of-Systems (SoS).
- Increasing complexity of the interaction between the (digital) system and the (analog) physical environment in which it has to operate leading to Cyber-Physical Systems of Systems (CPSoS).

System complexity expands the number of design alternatives. Therefore, it is crucial to assist the system architect in the identification of the most suitable solutions already at the early phases of the design process before time and effort are invested in the implementation, integration and testing [2]. Moreover, selecting the hardware platform from the very beginning allows the development of the hardware architecture in parallel with the development of the software architecture, thus accelerating the global development process. The challenge for the industry is to avoid over-dimensioning the required computing resources (with impact on cost and energy) as well as under-dimensioning them (impact on performance and redesign cost).

Model-Driven Software Engineering (MDSE) has proven to be a powerful approach to deal with the increasing complexity of software development [3]. It can be adapted to different design contexts and

* Corresponding author.
E-mail address: villar@teisa.unican.es (E. Villar).

https://doi.org/10.1016/j.micpro.2020.103244
Received 18 November 2019; Received in revised form 17 August 2020; Accepted 19 August 2020
Available online 29 August 2020

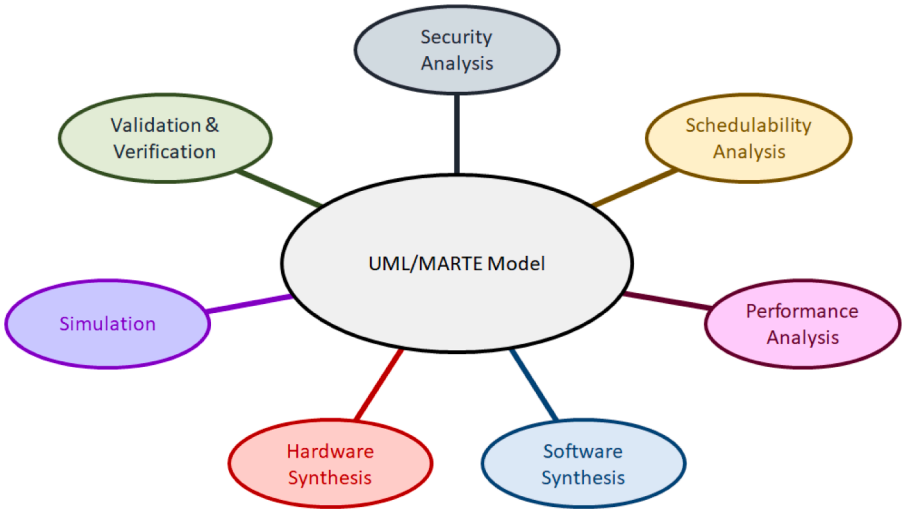

Fig. 1. The S3D framework.

domains, being compatible with methodologies like Agile [4] and DevOps [5]. The abstraction, interoperability, and reusability capabilities of MDSE become especially relevant in order to model complex services built on distributed, heterogeneous embedded devices as commented above. The advantages of MDSE can be applied to system engineering as well.

System modeling is an essential aspect of any system engineering methodology. This is especially important in Model-Driven Engineering (MDE). MDE demands a sound, comprehensive modeling methodology in order to capture all the relevant information about the system being designed. Thus, in order to address the demand for more powerful modeling methods, new methodologies are continuously being proposed. Currently, the challenges posed by the increasing complexity of CPSoSs require of more powerful, generic, reusable, platform-independent modeling methods able to cope with the complete service under design. From the model, it should be possible to analyze, explore different solutions and optimize the system towards satisfying all the functional and non-functional requirements such as speed, throughputs, power consumption, security, safety, etc. Finally, once the system engineer is convinced that the system design is valid, it is possible to synthesize the SW to be executed on the different computing resources and, eventually the application-specific HW required.

The tendency in the last years has been towards a specialization of SW development methods and languages to specific domains, leading to a diversity of Domain-Specific Languages (DSLs) [3]. Nevertheless, the use of these DLSs usually creates closed areas, since it is quite difficult to integrate under one DSL the developments done in others, limiting reusability and composability. To solve this problem, this paper proposes, first, the use of a standard modeling language, and then, to specify the system components in such a way that, they can be easily adapted for its integration under different environments, communication mechanism and/or models of computation.

To achieve this goal, this paper proposes the use of UML/MARTE (Modeling and Analysis of Real-Time and Embedded systems), the standard proposed by the OMG for the modeling and analysis of real-time and embedded systems. UML/MARTE covers both system engineering, by the Generic Resource Modeling (GRM) and the Generic Component Modeling (GCM) chapters, and software engineering, by the High-Level Application Modeling (HLAM) chapter and the Software Resource Modeling (SRM) section of the Detailed Resource Modeling (DRM) chapter. In addition, UML/MARTE covers architectural mapping and design-space exploration by supporting the description of the computing architecture by the (Detailed) Hardware Resource Modeling (HRM) of the Detailed Resource Modeling (DRM) chapter [6].

One of the main ways to improve design productivity is to keep to a minimum the need to develop new components from scratch but using them repeatedly from one project to the other [7]. This affects several versions of the same product in a product-line, the same product implemented on several HW/SW execution platforms as well as the maintenance and updating of a product already in field operation.

As a result, in this paper, S3D, a Single-Source System modeling and Design framework is proposed able to support an efficient Cyber-Physical Systems of Systems (CPSoS) modeling. The starting point is UML/MARTE as it allows to model functionality at different abstraction levels but also the execution platform on which this functionality can be mapped. In a Single-Source approach [4], all the relevant information about the system being designed is centralized in a single model. The rationale behind this approach comes from the fact that modeling is costly and error-prone. The main goal of the S3D approach is to minimize the modeling effort as much as possible. In order to facilitate capturing all the relevant information about the system for different purposes in a coherent, accessible and understandable way, the information is organized in views. Each view encloses all the required information about a particular aspect of the system. As each view is orthogonal to the others, they support separation of concerns, which is an important principle for designing high quality software systems and is both applied in the Model-Driven Architecture (MDA) [8] and Aspect-Oriented Software Development (AOSD) [9]. An essential novelty in S3D is the description of component functionality. Instead of making use of the mechanisms provided by UML for that purpose like state or activity diagrams, S3D associates to each component its functional code in a programming language (i.e. C++). Components may be associated to several codes in different languages so that the most appropriate for a particular computing resource or application can be selected. S3D does not impose any restriction to the code in the file which is a user's responsibility. The only strict condition is to ensure that the code is platform-independent so that it can be compiled and executed on any computing platform. The goal of the S3D framework is the automatic generation of the simulation and analysis models from the same source, taking into account the architectural and mapping decisions as well as the communication and synchronization properties among components defined in the UML/MARTE model.

From the central repository, different tools can be used in order to perform the different design tasks such as verification, simulation, performance analysis, schedulability analysis, etc. Finally, when the design is considered correct, satisfying all the functional and non-functional constraints, the code to be deployed on the different computational nodes of the distributed platform is automatically generated, as shown

in Fig. 1.

From the same single-source model, Model to Model (M2M) and Model to Text (M2T) generators extract the information required by each design tool. Once the design is considered as correct, the code to be compiled on each computing resource or synthesized (using behavioral synthesis tools), is extracted from the model. In this paper, the focus is put on the modeling methodology able to support such a framework. Additional information about S3D and its associated design tools can be found in [10].

The structure of the paper is the following. First, a description of the state of the art regarding the modeling approaches currently available will be provided. A discussion of the problem statement and the main requirements that the methodology should satisfy is provided in the next section. Then, the S3D modeling methodology is detailed. First, for the sake of completeness, the fundamental concepts supporting the methodology are described. Then, the main improvements to current common practices towards reusability are highlighted. Finally, the modeling methodology is assessed on a complex use case, a Flight Management System (FMS).

2. State of the art

Since the 90's, Object-Oriented Modeling is the dominant SW modeling and design methodology [11]. In Object-Oriented Modeling (OOM), the system is conceived as a collection of objects. Objects are instantiation of classes, which encapsulate data and methods. No restrictions are put on the way the objects interact among them, either by calling methods from other objects or global and static variables. Concurrency is not made explicit. Thus, a class may trigger a large number of threads or may be a passive unit implementing methods to be called from external objects. As the main communication mechanism is the function call, active threads jump from one object to the other freely. This makes very difficult to analyze the actual behavior of the system being modeled. As each object may interact with any other, understanding the active threads in the system is not easy. In the same way, apart from inheritance, hierarchy is not visible in many cases. This makes OOM hardly scalable and reusable. The problems derived from concurrency in OOM have been highlighted [12].

In Actor-Oriented Modeling (AOM), the system is conceived as a collection of concurrent components called actors [13][14]. Actors encapsulate data and functionality and interact each other through predefined communication patterns, which may lead to concrete Models of Computation [15]. Actor-Oriented modeling intends to highlight 'concurrency, temporal properties, and assumptions and guarantees in the face of dynamic system structure'. Although it is more restrictive than OOM, the benefits that AOM provides justify its use. Examples of AOM frameworks are Simulink [16], Labview [17], Modelica[18] and Ptolemy [19].

In Component-Based System Modeling (CBSM), the system is designed by hierarchically dividing its structure in components interacting among them and with the environment. The other way around, in a bottom-up approach, the system is built up from predesign components [20]. A component is a concept in the middle of an object and an actor. CBSM imposes conditions to the objects in order to be considered as components. While an object does not have any restriction in the way it interacts with other objects, a component encapsulates functionality and interact with other components using explicit communication interfaces. Therefore, CBSM can support OOM directly as components are objects and can be used as such when the restrictions to the communication and synchronization mechanisms among components are relaxed. In the opposite direction, when the internal functionality of the component has to satisfy concrete restrictions, it becomes an actor.

Model-Driven Software Engineering is widely used in order to cope with the increasing complexity of software development. Several commercial tools are available. One of the most popular is Matlab/Simulink [21]. Although Matlab/Simulink facilitates the modeling and analysis of complex systems, its simulation efficiency might be an important disadvantage. Being based on a single Model of Computation and Communication (MoCC) is another limitation. CoFluent is other commercial tool extended to model IoT systems [22]. Although supporting more interaction models that Matlab/Simulink, it is also limited in the way components may interact among them. On the contrary, Ptolemy is an open-source software framework supporting actor-oriented design under any MoC [19]. The flexibility Ptolemy provides in mixing different MoCs goes hand in hand with restrictions in modeling as the action code has to be encapsulated inside the Java infrastructure combining the different 'Directors', that is, the simulation engines for each specific MoC.

The communication and synchronization mechanisms among components are essential to define the underlying MoC on which the system is based. Software connectors are first class architectural elements that reflect the specific features of interactions among components in a system [23]. They are associated to a protocol and an implementation [24]. In some cases, specific UML profiles are proposed to describe the SW architecture of components and connectors. The latter are described using a specific stereotype very similar to the component [25]. Being an architectural element, connectors can be composed to build more complex interaction mechanisms among components [20]. This solution makes it more difficult to model the SW architecture and makes it more rigid as any change in the interaction between two components requires replacing the connector in the composite diagram. A taxonomy for connectors has been proposed but the result is difficult to handle as the variety of connector types considered is very large [26]. In order to specify the behavior of connectors, they can be handle using the same mechanisms as for behavioral types [27]. Software connector may strongly benefit from the flexibility provided in this paper in defining different MoCs.

The Unified Modeling Language (UML) provides a standard, graphical-based formalism for capturing system models. UML is very flexible but lacks the semantical content required in most application domains. The fUML subset details the simulation semantics of the model but it does not add the concepts needed in a particular domain [28]. As a consequence, the tendency has been towards a proliferation of DSLs [3]. Metamorph is a good example of a modeling framework able to combine different DSLs. The key feature in achieving this goal is the 'connector'. The connector is language-specific (i.e. Modelica connector or Spice connector). Heterogeneous simulation is achieved by combining different simulation engines under the global control of a Master, in a similar way as Ptolemy [29]. Addressing IoT heterogeneity by combining different DSLs has been proposed in [30]. This is the easiest way but strongly restrict design-space exploration as changing a component modeled in one DSL from one domain to another would require to model again the component in a new DSL, eventually from scratch. Even in the case the transformation is automatic, some domain-specific information may be lost.

An alternative to UML for Model-Driven Design (MDD) of systems (and SoS) is the Architectural Analysis and Design language (AADL) [31]. As the language is based on fundamental computing engineering concepts for both the functional architecture and the HW/SW computing platform, AADL could be used in many different domains. Nevertheless, its current use is limited to mixed-critical systems in the automotive and aeronautic domains. An AADL model contains component types and implementation with their interfaces, subcomponents, and other properties. It defines the system in a hierarchical manner, with a top component called the root system and other component categories are grouped into three clusters: hardware, software, and hybrid. Components communicate among them connections. AADL supports three types of connections. A port connection represents the transfer of data and control between two concurrently executing components, i.e., between two thread components, or between a thread and a processor or device. Parameter connections are an abstraction for the flow of data through the parameters of a sequence of subprogram calls. Access

connections designate access to shared data components by concurrently executing threads, or by subprograms executing within a thread. As a consequence, modeling a particular communication mechanism requires using different types of connections with different properties which makes difficult to change the underlying MoCC and explore different solutions [32].

Several modeling environments such as Modelio [33] and Papyrus [34] support UML/MARTE, the standard DSL for real-time and embedded systems. The standard defines the basic concepts required to support real-time and embedded systems, a first specialization of this core package to support pure modeling of applications (e.g. hardware and software platform modeling) and a second specialization to support quantitative analysis of UML2 models, specially schedulability and performance analysis. Nevertheless, although MARTE provides the required modeling elements but does not define how to use them. As a consequence, several modeling methodologies have been proposed [35]. One of them is Time4Sys, a framework developed as a Polarsys plugin with the objective to bridge design and analysis tools without changing the development framework. The main goal is timing analysis. It supports schedulability analysis based on 'Worst-Case Execution Times' (WCET). The tool also supports workload simulation assigning constant execution times to the tasks (subtasks) [36].

One of the first frameworks using MARTE for system modeling and design was Gaspard [37]. In Gaspard, components encapsulate tasks. The underlying MoC is data-flow. The main contribution of the paper was to take advantage of the Repetitive Structure Modeling (RSM) capabilities of the MARTE profile. The RSM features facilitates the description of iterative architectures in which the system is composed of a multi-dimensional structure of tiles of the same component. Three technologies are targeted for system analysis at different abstraction levels: functional level with synchronous languages, PVT level with SystemC, and RTL level with VHDL. The equivalence among these three different executable models is not explained.

Chess is one of the most complete UML/MARTE component-based modeling methodology [38][39]. As Gaspard, Chess also exploits the potential of UML/MARTE in describing the HW platform and, therefore, easy the modeling of different architectural mappings. Another important similitude is the solid implementation of separation of concerns by design views. Some SysML features are used for the modeling of requirements and for the system-level design. Functional and temporal properties and requirements of components can be specified in Chess following the FoReVer contract-based approach [40]. Although the original focus is embedded and real-time software, Chess can be applied to other domains as the semantical concepts supported are fundamental to system engineering and not domain-specific. Components interact among them through ports. The functions that a 'ClientServerPort' offers to, or requires from other components are declared in the interfaces associated to the port. The properties that the system engineer can assign to the interface functions in order to specify its execution only refer to its periodicity (sporadic or periodic) and its protection (sequential or guarded). In order to facilitate the modeling of data-flow systems, the 'flowport' is also supported. Being, apparently, an advantage, the 'flowport' enforces a concrete communication mechanism, which limits the flexibility and reusability of the system model. Another, more recent UML-based modeling and design framework is Hepsycode. Its main limitation is that it only supports a single MoC, the CPS [41].

Reusability has been always an implicit consequence of MDD. The distinction between the Platform-Independent Model (PIM) and the Platform-Specific Model (PSM) made by Model-Driven Architecture [7] facilitates reusability. If the MDD methodology is component-based, then reusability is even easier [42]. Nevertheless, reusability does not come for free and additional measures have to be taken in order to ensure it [43]. There is a lack for system modeling methodologies supporting reusability as a first-class goal. In most cases, only general recommendations are found in bibliography [44]. So, the use of repositories of models is mentioned. Nevertheless, the structure and contents that these repositories (libraries) should have, are not described [42-44].

The UML/MARTE background modeling methodology on which this paper is based was presented in [35]. Its fundamentals were similar to Chess. Both are good examples of generic, reusable, platform-independent modeling methodologies. They have proven to support the modeling of embedded systems. Nevertheless, when dealing with services implemented on CPSoSs, these MDD frameworks need to be extended with more powerful modeling methods. In a CPS, the embedded system has to operate inside a physical environment ruled by strict physical equations. So, for instance, in the FMS used as example in the paper, the position of the airplane at each point in time is determined by the impulse produced by the engines, the mass of the airplane and the aerodynamics of its movement. The importance of accurate timing analysis in CPSoS design has been highlighted [45,46]. Time accuracy and granularity evolve along the design process. From an untimed model of both the environment and the system in which data in the environment are generated and consumed at certain implicit rates, more detailed models can be derived. For the environment, SystemC is a good choice. Its analog extension may facilitate the modeling of physical processes [47]. Regarding the temporal behavior of the system, it can be untimed at the beginning, timed with estimated workloads during system analysis and optimization and finally, modeled accurately using host-compiled simulation [2] or virtualization. In all the cases, the system model has to be generated from the functional code by the M2M and M2T generators commented above. During system optimization, the functional code in components may require to be replaced by optimized versions for specific targets (i.e a GPU requiring OpenCL instead of C++) or its parallelization in order to exploit the parallelism of NoC-based multi-core platforms [48,49] and MPSoCs [50].

In this paper, the improvements built on top of S3D will be described. The main goal of these improvements is reusability. The structure of the paper is the following. In Section 3, an overview of the methodology is presented. This will allow the reader to better understand the technical contributions made. These contributions are detailed in Section 4. In Section 5, an industrial use case, a Flight Management System, is used to assess the methodology proposed and the advantages it brings in supporting a fast exploration of different architectural mappings and the selection of the most appropriate. Although the experiment corresponds as well to a design-space exploration example, the different architectural mappings may correspond to several versions of the same product, an up-date of a product or a new implementation of the product on a different platform. Finally, the main conclusions of the work are outlined.

3. Problem statement

As commented above, the main challenges to be faced by a CPSoS modeling methodology are the increasing complexity of the integrated circuits, its growing heterogeneity and the increasing complexity of the system behavior and structure, composed by a number of distributed embedded systems, eventually connected to the cloud. From the analysis of the state of the art, it is possible to conclude that there is a lack of powerful-enough system modeling methodologies able to scale to cyber-physical systems composed by a variety of embedded systems.

In order to minimize the modeling effort and to reduce the number of errors, the modeling methodology should be simple, easy to understand and easy to be applied to different domains. As an additional characteristic towards simplicity and understandability, the number of fundamental modeling primitives should be as reduced as possible.

However, even with a simple methodology, modeling a complex system can be a time-consuming task. To minimize that issue, out first proposal is to ensure that the same system model can be used for multiple purposes. That way, the modeling overhead impact along the whole design process is limited. For such purpose, we have adapted our modeling methodology to be able to generate models that can be used as inputs for many tools, such as eSSYN for code generation [51], VIPPE for

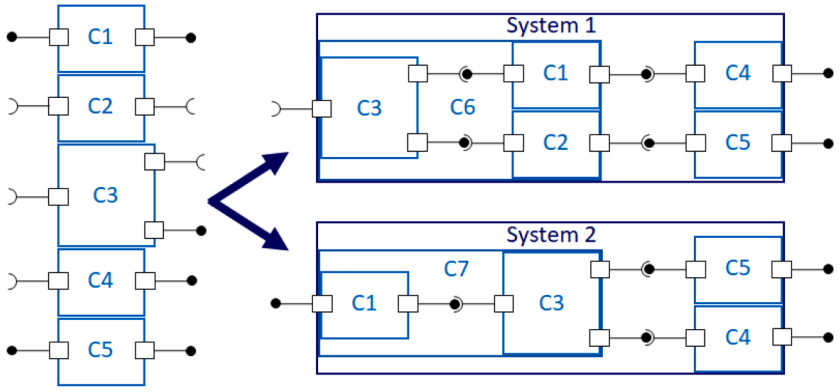

Fig. 2. Different systems composed from the same components.

simulation and performance evaluation [52] or MAST for schedulability analysis [53]. As a result, the system model can be used as a single source for different activities.

The single-source modeling approach followed by S3D supports capturing all the relevant information in a single site thus avoiding duplication of design information. If a piece of information is relevant for the system under design, it should be part of the model. The single-source approach ensures that any piece of information is where the tools needing it will find it and linked to those other parts of the model to which it is related.

3.1. Reusability

Generating a complete model of a large system (i.e. a CPSoS) is a time-consuming task. This paper focuses on proposing solutions capable of reduce this effort by improving model reuse, as shown in Section 5. These improvements are presented for the S3D modeling methodology, but they could be applied to other methodologies, especially due to the fact that they are based on standards, not on specific profiles, as described below.

Typically, the process of creating a UML model involves, first, creating models of the individual elements of the system (e.g. components) and then, combining these elements to specify the system. It is usual to spend more time in the modeling of the components than in the modeling of their interactions, due to the different parameters and details to be configure within the component models. Thus, minimizing this time will represent an important reduction of the overall modeling time. To do so, as in many other design activities, one of the main ways to improve design productivity is to keep to a minimum the need to develop new components from scratch but using them repeatedly from one project to the other [7].

However, there are several problems when trying to reuse components, that will be addressed in this paper. The first problem is that, fundamentally, MDD is a top-down approach. The functional model of the system is developed by its hierarchical partition until simple enough components are found and developed from scratch. The model is analyzed and optimized until a valid model is ready for development. On the contrary, reusability is a bottom-up approach in which previously developed components are used again in order to build a new system. Thus, our first idea is to change this view. The proposal is to develop a library or repository of component models as complete as possible, valid to be reused from one project to another.

But reusing a component from one project to another is not so simple. A second problem is that the reuse ratio can change a lot from one design to another. If a company is developing an improved version of a previous project, the new design can potentially reuse a lot of components from the previous version. However, if the company is developing a design for a completely new area, reuse can be minimal. In that context, it could be required to get third party components. However, this is only possible if standards are used. Modeling all the required details to support different tools with standard UML profiles is difficult, since standards are common and, thus, they do not offer exactly all the elements required by a specific tool provider. Typically, specific profiles are proposed to solve this problem, but they go against model reusability. Thus, an alternative based on standards is proposed in this paper. Specially, the paper will show how communication semantics can be modeled using the standard MARTE (Modeling and Analysis of Real Time and Embedded systems) profile of the OMG while ensuring component reusability.

Furthermore, to be reused, a component may require being integrated in a completely different structure compared with the design where it was originally developed (Fig. 2), services can be requested with different communication semantics, or the functionality has to be run in a completely different hardware platform. Achieve reusability under these circumstances requires the components to be completely independent from its implementation and use. To do so, first, component internal functionality must be completely platform independent. Secondly, another characteristic to be covered is composability. Components should be able to be connected each other without restrictions, whenever one provides the services the other requires. In this way, the modeling methodology should support a 'bottom-up' design methodology where sub-components are built up by the composition of simpler components, which, in the same way, can be the result of the composition of other simpler components as shown in Section 4.

Thus, the main goal of this paper is to propose solutions to improve component reuse, focusing on platform independency and communication independency, including independency on ports and communication semantics, including support for multiple Models of Computation and Communication (MoCCs). Especially, the example and results in Section 6 will demonstrate how the same functional components can change their models of computation with just change a few parameters in the component model, using automatic code generation to obtain implementation with different performance.

4. S3D: Single-source system modeling and design

For the sake of completeness and to present the improvements on reusability described in Section 5, a brief introduction to the S3D modeling methodology is required. As stated before, the S3D methodology has been proposed to enable designers to create system models that can be used as input for different tools at different steps of the design process.

The modeling methodology has been kept simple, based on fundamental system engineering concepts easy to be understood and easy to be applied to different domains. The number of modeling primitives have been reduced as much as possible, limiting them to ensure unambiguous identification of the fundamental concepts in the model. The methodology is Component-Based [48] and therefore, the main modeling primitives are the components, the required and provided

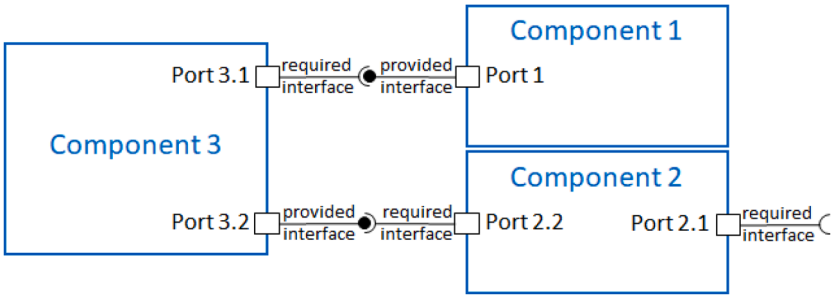

Fig. 3. A system with three components.

interfaces and the ports. Interfaces are basically lists of services offered or required, being services in S3D functions (defined by their name and arguments) for communication from/to other components. Ports define the communication capabilities of the component. Thus, each port indicate a list of interfaces, that is, a list of services, which are, all of them, required or provided. As it will be described in Section 5, ports will also define the communication and operation semantics associated to these services, such as synchronism or concurrency.

Additionally, to capture all the relevant information in a simple way, the system model is divided in 'views'. Each 'view' captures all the relevant information about a specific design concern (i.e. component characteristics, system functionality, the system verification, HW platform, etc.).

Following the Model-Driven Architecture (MDA), the design and verification views are divided in three groups, the Platform-Independent Model (PIM), the Platform Description Model (PDM) and the Platform-Specific Model (PSM). Nevertheless, the actual meaning of these models in S3D has been modified in order to integrate the information required by the single-source approach.

Additionally, although simple, the modeling methodology should be able to support the modeling and design of complex systems, even Cyber-Physical Systems of Systems (CPSoS). This would enable a holistic analysis of the complete service being supported by the SoS, thus allowing the right system-level architectural decisions.

4.1. Platform independent model

The PIM "exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms" [3]. Thus, it includes all these elements that are related to system functionality. These elements include the description of the functional components, the overall functional architecture and the model of the environment for functional verification. As no relation to any HW architecture is included, the model is completely platform-independent.

4.1.1. Application components

The fundamental modeling element in the methodology is the component. Components communicate among them through ports, and ports contain interfaces, which define the communication methods. The components either require communication methods (or services) trough required interfaces or offer communication methods (or services) through provided interfaces (Fig. 3).

Reusable components in the library can be <<RtUnit>> or <<PpUnit>> weather they are active, concurrent objects or passive ones. When, related information can be assigned, such as the maximum number of threads a 'RtUnit' can have active at a time. Additionally, 'RtUnit' components can have their own internal execution flows (not triggered by an external service call). So, a thread is generated executing the component method specified as the 'main' function, leading to component-level parallelism. Functional and extra-functional constraints may be imposed to the application components using appropriate constraint-specification languages such as OCL.

Functional components also include the specification on the functionality, while ports and interfaces specify the semantics of each communication. However, the definition of these elements is critical in terms of reusability. Thus, they will be described in detail in the next section.

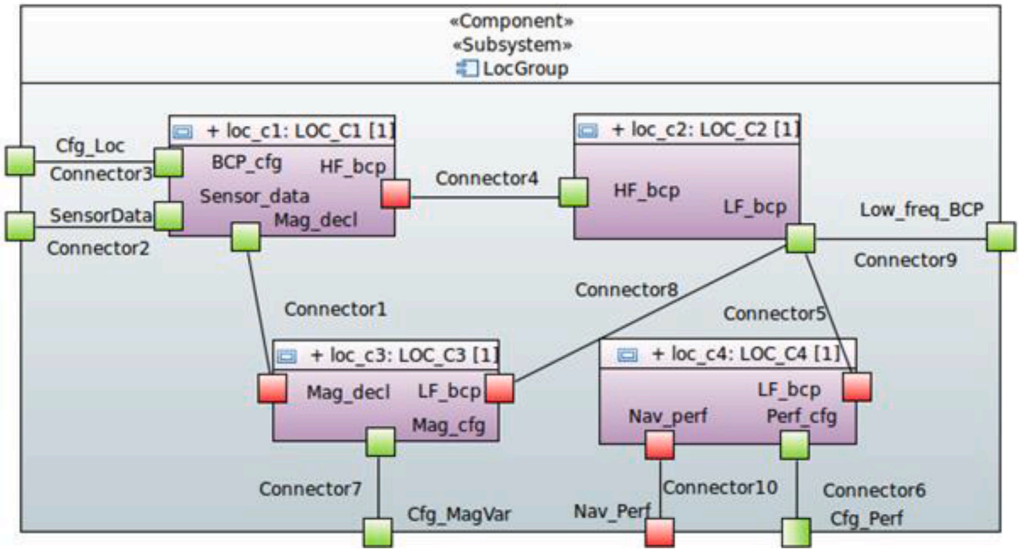

Fig. 4. The structural subsystem LocGroup.

4.1.2. System functionality and subsystems

The system is conceived as a hierarchical network of components. Once the components are specified, the system functionality is obtained as a composition of such application components connected each other through concrete, compatible ports and interfaces. Thus, the system is created following a client/server philosophy, being the client the component requiring a service, and the server the component providing this service. In order to identify a component as the system, the <<System>> stereotype is used.

Additionally, application components can be grouped together in subsystems. A subsystem is just a component that includes other components inside, resulting a hierarchical component. In order to identify a component as a subsystem, the <<Subsystem>> stereotype is used. A subsystem can be part of more complex subsystems. In this way, subsystems are essential to deal with the modeling of complex systems of systems. That solution enables modeling hierarchy. Then, when a problem is too complex, the main way to address its modeling is dividing it in smaller sub-components, which can be modeled independently. An example is shown in Fig. 4: the LocGroup component of the Flight Management System used in Section 6 is shown.

Hierarchy is supported both at the functional level and at the platform, as described below. At the functional level, the functional architecture can contain as many sub-systems as needed. Each sub-system, if complex enough, can be decomposed in as many lower-level sub-systems as required. In this way, modeling of systems of systems is supported.

4.2. Verification

Verification is an essential task to be performed all along the design process. Each time the functional end extra-functional constraints for the whole system, its application subsystems and each of the components are defined, black-box verification suites at the different granularity levels can be set-up. When the functional code for each component is ready, concrete test sequences ensuring the correct behavior of the components can be developed. Based on these component test suites, sub-system and finally, system verification can be carried out. As commented above, the time accuracy and granularity of the environment will be refined as the design process evolves, leading to several models of the 'test-bench'.

4.3. Platform-description model

In this section, the fundamental elements of the PDM will be described. The PDM captures all the information required to describe the HW/SW platform of computing resources used to execute the system functionality described in the PIM.

4.3.1. Network nodes

In order to deal with the modeling of very complex systems of systems (SoS), partition and hierarchy are essential mechanisms to be exploited. The SoS should be partitioned in parts (i.e. complete systems by themselves) which should be partitioned again hierarchically until the detailed computing platform can be described by its computing architecture of HW devices. These hierarchical parts are nodes connected each other through a network infrastructure. The network nodes play in the platform description the same role as the subsystems in the Platform Independent Model. The type of connector in the model is related with the kind of network used (i.e TCP/IP). The connection can be attributed with performance properties such as bandwidth, delay, jitter, etc. from which analytical or dynamic models can be derived [45].

4.3.2. Software platform

An essential element in any computing platform is the Operating System (OS), eventually, several of them when the computing platform is complex and heterogeneous enough. In some cases, when a system or a subsystem has real-time constraints, a Real-Time Operating System (RTOS) is required.

Apart from the OS, there is another Hardware-dependent Software (HdS) that has to be taken into account. Peripherals and, eventually, co-processors may require specific SW to implement the high-level interface services used in the PIM. This code will be represented in the software platform as a <<DeviceBroker>> realizing a certain connection. The driver of the device should be installed in the OS.

4.3.3. Silicon implementation

The MARTE 'HW_Physical' model represents hardware resources as physical components with details on their shape, size, power consumption, heat dissipation, and many other physical properties. In S3D, 'HW_Logical' entities, apart from 'HW_PLD' and 'HW_ASIC' can be mapped to physical entities indicating a design intention or decision. As an example, in Fig. 6: , the ARM-Duo and associated devices are to be implemented in a FPGA, the main memory will make use of a commercial chip (stereotyped as <<HwComponent>>) and the non-critical resources will be implemented in an ASIC. Based on this information, S3D will generate automatically all the information needed to feed the corresponding design flows.

4.4. Platform-specific model

The Platform-Specific Model (PSM) captures the implementation decisions taken during the design process. These decisions lead to the architectural mappings to be analyzed, compared, optimized and, finally, selected. Design decisions are expressed in UML with the 'abstraction' and 'allocate' relation between objects. A is allocated in B means that object A is to be implemented by object B. So, Figs. 7 and 8 shows the design decisions taken for the architectural mapping of the FMS application components to the HW-SW resources of the actual PDM. Therefore, the methodology is flexible enough to support the analysis and comparison of many different architectural solutions for the

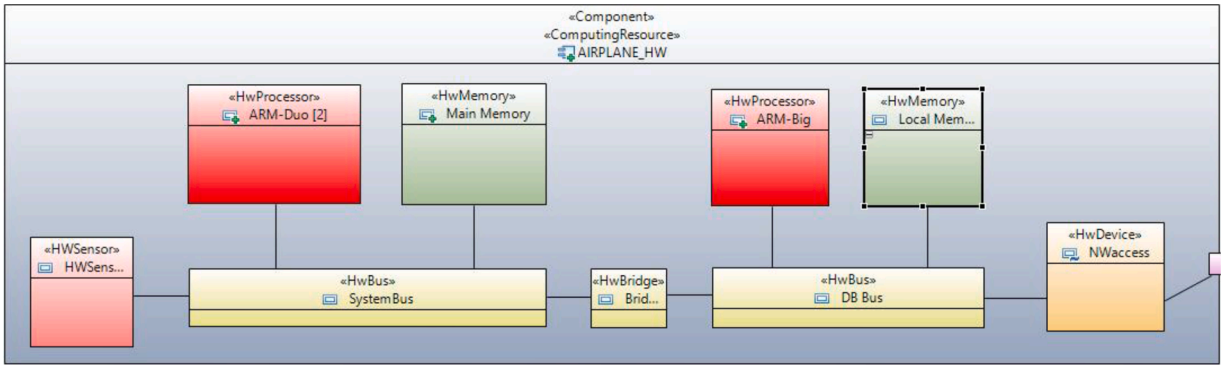

Fig. 5. HW architecture for the "Airplane_HW" Node.

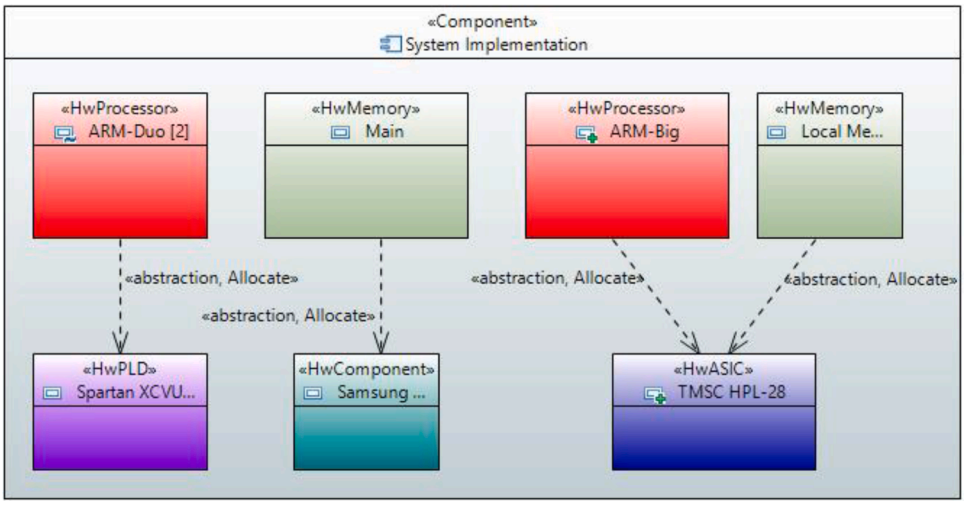

Fig. 6. HW implementation.

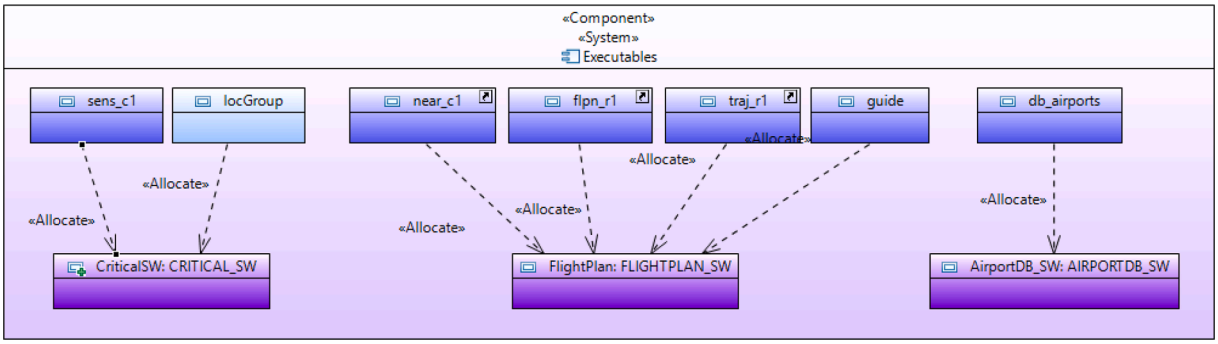

Fig. 7. Figure of functional components to executables.

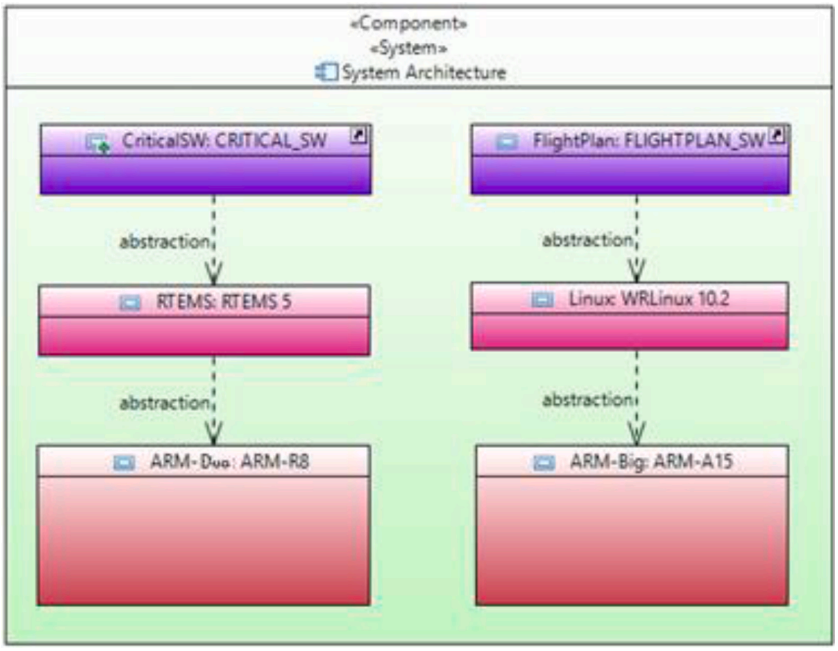

Fig. 8. Architectural mapping of FMSR components.

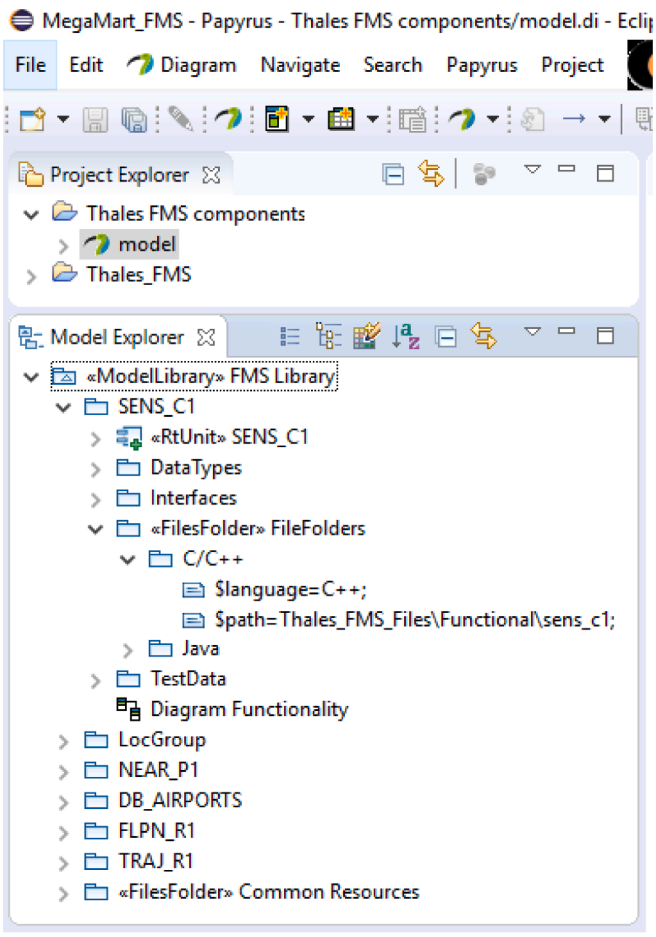

Fig. 9. An S3D reusable library.

implementation of a complex system. The system architect should be able to explore as many different architectural mappings as needed, that is, decisions about which computational resource should execute each functional component, with minimal effort. With just modifying the arrows in the view, the concrete analysis model of a particular architectural mapping is generated automatically [2].

4.4.1. *Memory spaces*

Depending on the characteristics of the computing platform, the application mapped on it may be implemented in only one process (an executable) or several. Each executable process will share the computing resources with the other processes but in its own memory space. Without this information, it is not possible to generate the application code. This is the reason why the functional components are mapped to memory spaces. Only when the complete system is simple enough this intermediate layer can be removed. Fig. 7 shows the implementation of the FMS in two executables, 'CriticalSW' and 'FlightPlan'. The airports database is to be implemented in a third executable.

4.4.2. *Architectural mapping*

Finally, memory spaces, SW resources such as operating systems and HW processors are mapped in the architectural mapping. In Fig. 8, the memory spaces (processes) defined in Fig. 7 are mapped to the operating systems, and those to the HW processors inlcuded in Fig. 5.

5. Improved reuse capabilities

In this section, the new features included in S3D in order to address the modeling requirements stated in Section 3 to improve modeling reusability, will be highlighted.

5.1. *Library-based methodology*

One of the main improvements provided by S3D is supporting bottom-up design approaches. In bottom-up design, the system is built as a hierarchical collection of sub-systems built-up from already characterized components stored in one or several libraries. In some cases, the components may come from the adaptation of legacy code from previous projects or they can be provided by third parties. Thus, any new project may reuse the components of the libraries or repositories linked to it. This point is critical, since without a proper reuse mechanism, the other proposals done to improve reusability will be useless.

These libraries contain complete functional component models, including all the relevant information about them such as the data types received or returned in the provided/required services, interfaces, properties, functional and verification codes, etc. Fig. 10 shows the basic structure of a library. Libraries can also include subsystems and components used to describe the platform such as processors or peripherals, including specific information, such as the characteristics of the caches or the ISA for processors.

It is worth mentioning that the functional model of a system is very similar to a sub-system in a library. Thus, once finished, including a system design as a new sub-system in a library is straightforward.

5.2. *Flexibility: Generic components*

A second proposal to maximize reusability and flexibility, is that components in the library are generic components. These generic components are specified considering a complete separation between functionality and communication in order to eliminate restrictions on their future use. Component functionality typically defines what the component does, and it is independently on where or how it will be used. However, communication is completely dependent on utilization details: target infrastructure, environment, surrounding components, etc. Thus, generic components are components that only include information about functionality and functional interfaces but not about ports or communication semantics. This latter information will be added in the model at the instantiation points, letting the automatic code infrastructure integrated in the S3D framework to generate the corresponding glue codes. That way, reusability and composability will be granted.

Following these ideas, the model of each abstract component follows the structure presented in Fig. 10. It includes:

- The UML component itself stereotyped as <<RtUnit>> or <<PpUnit>> depending it is an active component or a passive one,
- Data types and interfaces,
- Functional and verification code.

These generic components are transformed in application components when used in a system. These application components inherit all the information from the generic components of the library, only requiring specifying the ports, which depends on the system connections. This is shown in Fig. 10: .

In their most abstract form, the only external information about such elements is the services (functions) they provide and/or require, as shown in Fig. 12 for SENS_C1_G. Thus, the required interface of a structured component lists all the services that the component requires from other components or the environment. The provided interface lists all the services that the component offers to other components or the environment. The fact that the structured components do not specify which, and in what way the required/provided services are grouped in interfaces and exposed externally, maximizes the reusability of these components. This is the reason why abstract components does not include ports. The communication and computation semantics under which each service will be used, will be defined when applying the

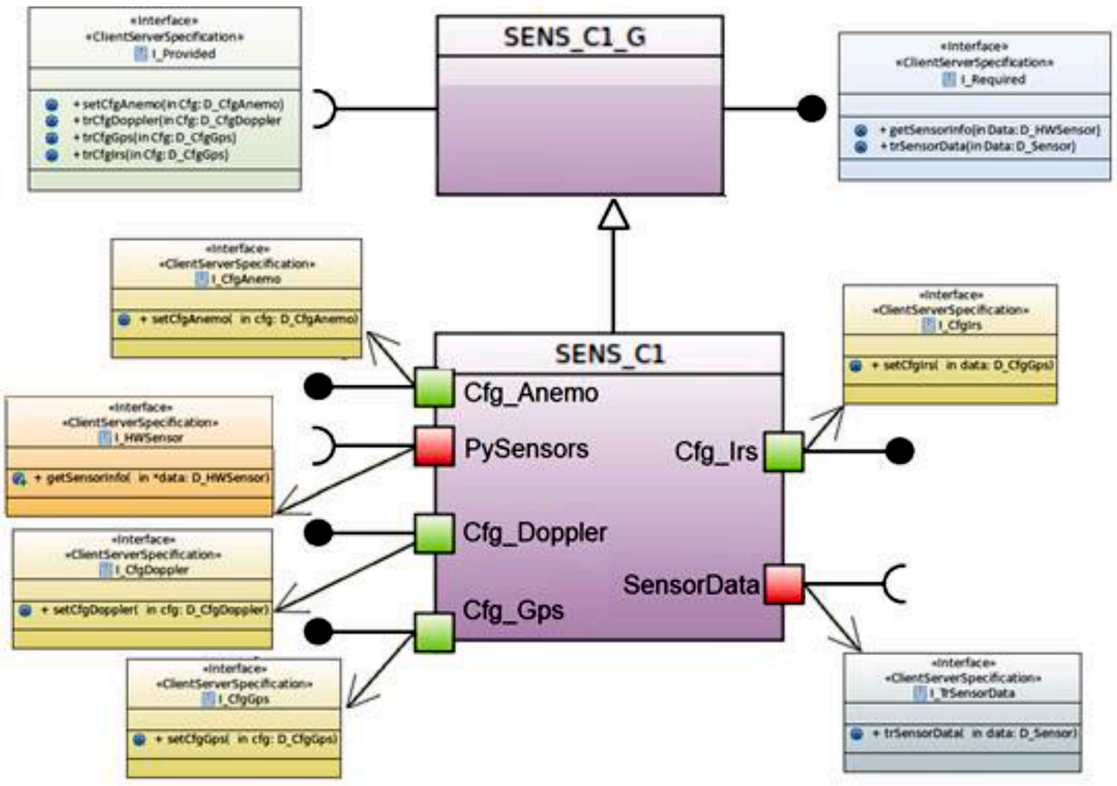

Fig. 10. SENS_C1 as a generic and as an inherited, application component.

component to a specific design, setting the ports required to specify these semantics.

Moreover, component models also include complete information about their internal functionality and provided services. In principal, no restrictions are imposed to the way the behavior is specified. Component-based design methodologies only impose restrictions on how the components interact among them, that is, through well-defined interfaces [54]. However, as a preferred solution, each component will be linked to the file where its structured data and behavior is specified in the action language used, in our case, C++, as described next.

5.2.1. *Descriptionof generic components' functionality*

The internal behavior of the component should be clearly documented and eventually, formally described. For that purpose, the main restriction is that the functionality must be described in a platform-independent way. In order to be platform-independent, the code should not include any system call or Hardware-dependent Software (HdS), Only once mapped to a concrete computing resource, the corresponding platform specific code including the required middleware, input-output access code and system calls is automatically generated by S3D by SW(HW) synthesis. The use of platform-independent code is important since any piece of code making use of any of these facilities limits its future reusability and goes against the separation between PIM, PDM and PSM. Moreover, decoupling component model from component implementation bring additional flexibility as the analysis tools may explore different implementation alternatives depending on the concrete architectural mapping decided.

Programming languages, databases, user interfaces, middleware solutions etc. are not considered part of the platform whenever they can be ported from one platform to another, thus supporting the execution of SW in several platforms. As a result, S3D uses programming languages (i.e. C++, OpenCL, Java, etc.) as action languages in order to describe the functionality.

Therefore, if an instance of the component is mapped to a CPU, the C++ or Java code might be the most appropriate, but, if it is mapped on a GPU, an OpenCL code may be preferable instead. However, obtaining a valid code from a generic functional description is beyond the current capabilities of S3D. Thus, when relevant, the new component may come with its implementation on a certain execution platform. That way the code is stored minimizing the effort of mapping the component in this resource in a future use. In fact, the automatic glue-code and makefile generator of S3D selects the platform specific code instead of the platform-independent code depending on the mapping of the component in the PSM.

Although in S3D the fundamental object is the hierarchical component, and, as such, it is a Component-Based System Modeling (CBSM) framework, S3D can support other system modelingparadigms like Object-Oriented Modeling (OOM) or Actor-Oriented Modeling (AOM) in a uniform and unified way. Code flexibility is an important feature in this sense as by relaxing the programing constraints, the CBSM becomes OOM. In the opposite direction, by constraining the internal behavior of the component, CBSM becomes AOM.

It is worth mentioning that, in the most general case, the generic components lack of any temporal information. This expands its reusability. When a generic component is instantiated, its instance can be annotated with the timing constraints to be satisfied by its implementation. This is why the functional code should lack of any temporal information. This information should be added to the simulation/analysis model depending on the temporal model used (i.e. estimated workloads or timed models based on host-compiled simulation or virtualization).

5.3. *Interface modeling in UML/MARTE*

Additional flexibility in the model is achieved by defining the properties of the functions in the provided/required interfaces. The goal is to provide the system engineer with a flexible modeling infrastructure able to support different system engineering methodologies. In some

cases, these properties my affect partly the programming of the component.

In the most general case, a required interface will call an interface function (service):

$$InterfaceFunction(X1, \ldots Xn, Z1, \ldots Zm);$$

where Xi are the input variables to the function and Zi are the output variables. Output variables are those that are changed as a consequence of the execution of the function. Input variables are those that may affect the final value of an output. Input variables can be passed by values or by reference. Output variables can only be passed by reference. An actual parameter can be used as input and output to a function simultaneously: $Xi \equiv Zj$

For example, consider a pair of components. The first one, acting as a client, request the service "checkMovement" to provide the GPS position and the movement angle of the plane. Other component, acting as a server, offers this service and, when executed, it receives this information and checks if it is correct conform the defined trajectory. Additionally, this information is stored in the server, which internally updates its status with the new GPS and angle.

According to our proposal, the previous code describes a fixed functionality, but supports multiple semantics, which are not defined yet. If "checkMovement" is called as a standard function call, the client will wait until the server executes the function, while the "main" function of the server will update the flight plan in an unsafe way, since "currentGPS" is used unprotected. However, this is not the only possibility. Modifying communication semantics in the model, and adding an automatically generated glue code in the middle, it could be possible to define that the service call is asynchronous, and thus, the client does not wait to the server. In this case, the client could continue its execution and call "checkMovement" again before the previous call has finished. Then, the new service call could be executed concurrently with the previous call, or not. If both run in concurrently, the second call could finish before the first one, and at the end, "currentGPS" and "currentAngle" would not contain the last value, but an old one. Then, let's suppose we select to disable parallelism. In that case, we should define if the second call should wait until the first one finishes, and the second call can execute; or if the call is cancelled as it cannot be executed immediately; or if the call should be stored in a buffer and executed when possible. In the latter case, we should define what to do when the buffer is already full; or if it is acceptable to potentially wait a long time blocked or we should put a timeout, etc.

On the server side, we could put a synchronization point between the service and the main function to protect the data, or just force the function to execute an iteration of the main function only when new data arrives, or depending on a time trigger, etc. It will depend, for example, on if we consider that every new GPS value should provoke an execution of "updateFlightplan" or if we only require a sufficient refresh ratio.

Thus, this example shows how channel semantics and mappings affect parallelism. As described in 4.1.1, in <<RTUnits>>, a thread is generated to execute the component method specified as the 'main' function, which results in a first level of parallelism. Then, when a service is executed, it can be executed in the client thread, if the call is local and sequential. Otherwise, a new thread is created in the server. Additionally, other channel semantics can limit the number of threads executing in parallel, as described below. The automatically created glue code, which is located in between clients and servers, oversees creating and controlling this parallelism, depending on the semantics described in the UML model. This code uses the services of the underlying OS for that purpose (e.g. POSIX threads, semaphores and mutexes).

Summarizing, multiple communication semantics can be implemented supporting the same code, resulting in different computation behaviors, some valid and others maybe not, with potentially different side effects on the rest of the system, and with different performance results. Thus, our proposal is that the library components should be initially specified without this information, which should be defined when the component is used in a design, obtaining an implementation according to the semantics from automatic code generation. This automation will also enable the designer to evaluate the alternatives he/she considers relevant without no design effort.

In order to improve reusability, the attributes describing communication semantics have to be modeled in a standard way. To do so, this paper proposes the use of the standard MARTE profile. However, using the profile for that purpose can be a complex task. In that way, we propose using attributes associated to the service, and to the provided and the required interfaces. Some of these elements will also enable describing different Models of Computation and Communication (MoCCs), as shown in next subsection.

5.3.1. Properties of the services of the interface

Each interface function (service) is stereotyped as a Real-Time service (RtService). Its properties are defined by the following attributes:

- The enumeration 'ConcurrencyKind' of the '**concPolicy**' attribute [0..1], which, in S3D, define relationships among services of the same component. The 'ConcurrencyKind' enumeration has three possible values:
- **reader**. The execution of the service has no side effects but can suffer side effects from other services. Consequently, the service can be provided concurrently to any other reader service (with the concurrency limit defined by the **srPoolSize** attribute of the corresponding component), but not with a writer service.
- **writer**. The execution of the service may have side effects. Consequently, once the service is provided any call to any other reader or writer service should be blocked.
- **parallel**. The service can be provided concurrently (with the concurrency limit defined by the **srPoolSize** attribute of the corresponding component) as it does not provoke or suffer side effects from other services.
- The enumeration 'CallConcurrencyKind' of the '**concurrency**' attribute [1], which , in S3D, define relationships among calls to the same service. Any MARTE RtService is an UML 'BehavioralFeature' and, as such, inherits the enumeration 'CallConcurrencyKind' of the attribute 'concurrency'. According to MARTE, the different values of the attribute are:
- **Sequential (S)**. When concurrency management mechanism is associated with the BehavioralFeature, concurrency conflicts may occur. Instances that invoke a 'BehavioralFeature' need to coordinate so that only one invocation to a target on any 'BehavioralFeature' occurs at once.
- **Guarded (G)**. Multiple invocations of a 'BehavioralFeature' that overlap in time may occur at one instance, but only one is allowed to start execution. The others are blocked until the performance of the currently executing 'BehavioralFeature' is completed.
- **Concurrent (C)**. Multiple invocations of a 'BehavioralFeature' that overlap in time may occur to one instance and all of them may proceed concurrently.

In S3D, sequential, is used to indicate that the code does not support concurrency (e.g. it is not reentrant), while guarded means that calls should not execute in parallel by any reason, such as the desired MoC (see next section), although it is technically possible. Thus, only G or C can be selected to define the operation semantics. S is in practice a restricted version of G that implies that the attribute cannot be changed to C in any case. As a result, S will not be considered when defined the MoCs in next section as it is not an election depending on the MoC.

As the 'concPolicy' attribute can also be applied to a PpUnit with the 'CallConcurrencyKind' enumeration, the value given to the PpUnit attribute will prevail to the value given to the attribute 'concurrency' of any RtService in any interface of the PpUnit.

- The enumeration 'ExecutionKind' of the '**exeKind**' attribute [0..1], which define when and where the service will be executed. The 'ExecutionKind' enumeration has three possible values:
- **deferred**. The call to the service is stored in the queue of the behavior attached to the service.
- **remoteImmediate**. The execution is performed immediately with the computing resource on which the called component has been mapped.
- **localImmediate**. The execution is performed immediately with the computing resource on which the calling component has been mapped. This possibility is not yet considered.

The enumeration 'SynchronizationKind' of the '**syncKind**' attribute [0..1], which defines the synchronity of the service. The 'SynchronizationKind' enumeration has four defined values:

- **synchronous**. The client waits for the end of the invoked behavior before continuing its own execution.
- **asynchronous**. The client does not wait for the end of the invoked behavior to continue its own execution.
- **delayedSynchronous**. The client continues to execute and will synchronize later when the invoked behavior returns a value.
- **rendezVous.** A behavior in the server waits for the client to start executing. In S3D, this means that it only forces the server to wait the client, not the client to wait for the server. If the client waits or not depends on the**notAttendedService**attribute of the client(see below).

5.3.2. Properties of the provided port

In case any of the RtServices of the interface is attributed with an execution kind 'deferred', then the provided port will provide a buffer to store the calls in the queue. The port will be stereotyped as 'StorageResource' and their properties defined by the following attribute:

- The integer attribute '**queueSize**' [0..1]. The integer value fixes the maximum size of the queue.
- The non-standard enumeration 'FullPoolPolicyKind' of the not standard '**fullPoolPolicy**' attribute [0..1], added in S3D to specify what to do when a FIFO is fool. The 'FullPoolPolicyKind' enumeration has five defined values:
- **block**. The call is not stored until a previous call is attended and a free position in the pool made available.
- **removeFirst**. The first call to be attended is removed and the new call stored.
- **removeLast**. The last call to be attended is removed and the new call stored.
- **flush**. All the previous calls are removed from the FIFO and the new call stored.
- **other**. Any other scheduling policy.

Both '**queueSize**' and '**fullPoolPolicy**' are applicable when '**exeKind'** is set to **"deferred".** The reason is that the corresponding FIFOs are only used to store the data incoming from clients that do not block themselves until the request is accepted.

5.3.3. Properties of the required port

When a required port calls a service, the call can be attended or not. The following attribute specifies the policy to follow in that case:

- The enumeration 'PoolMgtPolicyKind' of the not standard '**notAttendedService**' attribute [0..1], if the service request cannot be attended just when it arrives. The 'PoolMgtPolicyKind' enumeration has five defined values:
- **infiniteWait**. If the call is not attended, the client component waits indefinitely until the call is attended.
- **timedWait**. If the call is not attended, the client component waits for bound time until the call is attended. At the end of the waiting time, if the call is not attended the behavior is determined by the 'retry' attribute.
- **dynamic**. If the call is not attended, the client component continues execution.
- **exception**. If the call is not attended, the client component raises an exception.
- **other**. Any other policy. In S3D it is used to specify that the service also requires to meet an additional condition to be executed, being especially important in event-based MoCs.
- The attribute '**timeout**' [0..1] specifies the time until an event occurs, and in S3D operates together with the previous one. When '**timedWait'** is specified, this attribute indicates the maximum time to wait. When **'other'** is selected, it indicates the time advance requested for the next execution, as described in section 5.5.4. With other values it is not applicable.
- The integer attribute '**retry**' [1] = 0. The integer value fixes the number of times the client will repeat the call. If the call is not attended in any case, the client raises an exception which will determine the policy to follow. It is not used under '**infiniteWait'** value**.**

5.4. Definition of models of computation and communication with MARTE

Depending on the properties defining the services and the provided and the required ports, different programming models can be defined. However, the selection of the values of all these properties require deep knowledge of the effects and the resulting semantics. Additionally, selecting a combination of properties without this knowledge can lead to inconsistent or incompatible results. For example, it has no sense to specify a service as deferred and synchronous at the same time.

On the contrary, when a designer selects a set of properties for a communication between two components, it is because he has a previous idea of how the result must operate. Since most of the times these selections come from well-known computation mechanisms, an overview of some of the most common Models of Computation (MoC) and their specification following the proposed modeling methodology is presented next.

The use of properties in both in the communication (require/provide) for defining the MoC is a challenging issue, since the same component must provide several semantics, event at the same time if it is connected with more than one client. The idea of limiting functionality to platform independent codes and specifying all communication semantics in MARTE stereotypes enables the use of the components under multiple semantics, since the automatic code generation can generate the glue code handling these semantics from the model information. Additionally, it is important to note that the main attributes defining the MoC are defined by the provider, in the RTService, although almost all of the stereotypes defined above has some impact. This fact simplifies the implementation process, since a server providing a service to multiple clients only have to provide one MoC or MoCs of the same group.

The problem appears when a component offers services with different attributes, and thus, potentially under different models of computation. However, S3D methodology considers that the problem is not the different models of computation but the relationship of the services with the internal state of the component. If one service modifies the state and another service uses it, it can be required to protect their execution, independently of their model of computation. For that purposes, the attribute '**concPolicy**' of the RTService is used. Thus, the attribute '**concPolicy**' is not used on the definition of MoCs, but critical in order to ensure its correct operation.

The general division of MoCs depending on the RTService properties, are listed in the next table, since more specific details for each group of MoCs are described in the next subsections (Abbreviations of MoCs used

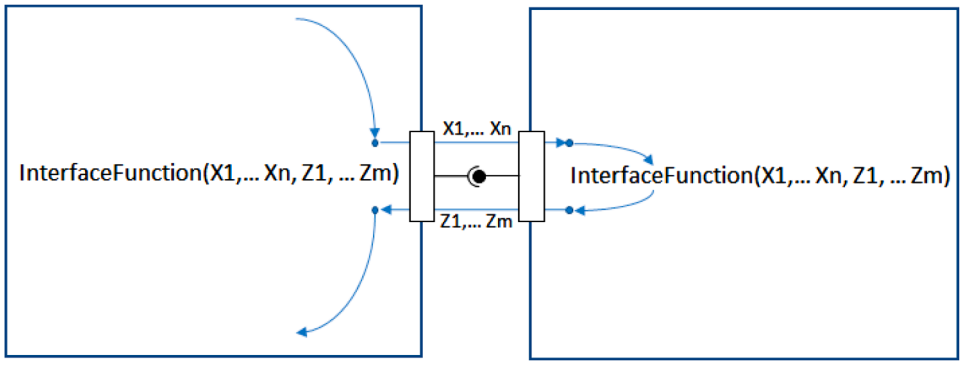

Fig. 11. RPC MoC.

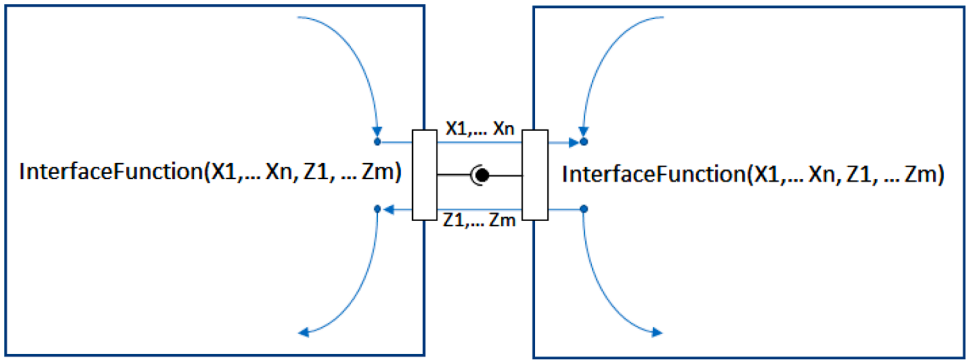

Fig. 12. Rendezvous MoC.

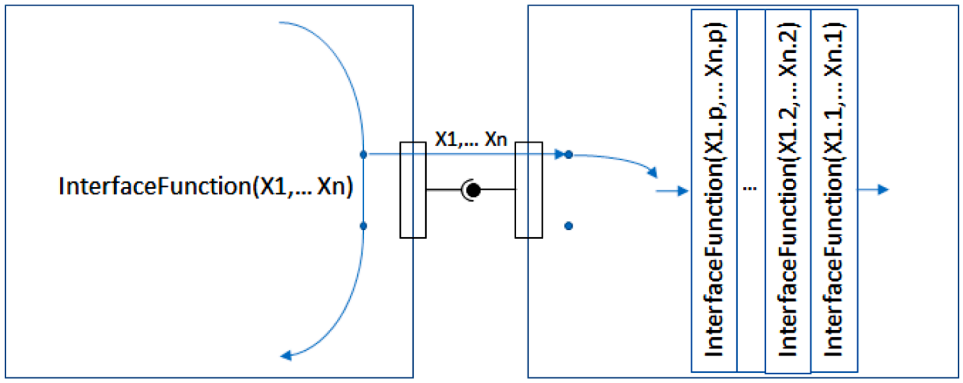

Fig. 13. Data flow MoC.

in the table are also described in next subsections).

5.4.1. Remote procedure call (RPC)/Remote method invocation (RMI)

A Remote Procedure Call (RPC), or its counterpart in object oriented languages, a Remote Method Invocation (RMI), are typically initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. While the server is processing the call, the client is usually blocked (it waits until the server has finished processing before resuming execution), as shown in Fig. 11.

However, certain RPC implementation also enables the client to send an asynchronous request to the server. As a result, both synchronous and asynchronous calls are possible. Additionally, in order to avoid deadlocks, a timeout can be defined.Table 2 shows the different alternatives. The RtService can be guarded or concurrent. In the second case, the server may attend several calls from the same component by parallel threads or several times the same service if repetitive calls are not filtered. Additionally, the attributes NotAttendedService, retry and syncKind define the exact RPC semantic, as described in [55].

5.4.2. Rendezvous (RV)

This is the fundamental communication/synchronization pattern for the Communicating Sequential Processes (CSP) MoC. In this case, the calling function requires the execution of the called function, which has to be executed by a main thread in the component providing the function:

The rendezvous ensures that two active tasks synchronize and interchange data at the same time. After the rendezvous, both threads are free to continue execution. In order to reduce the interaction time, the execution time of the called function should be minimized. In most cases, the function is just instrumental to interchange data, Xi in one direction and Zi in the opposite. In some cases in which the execution

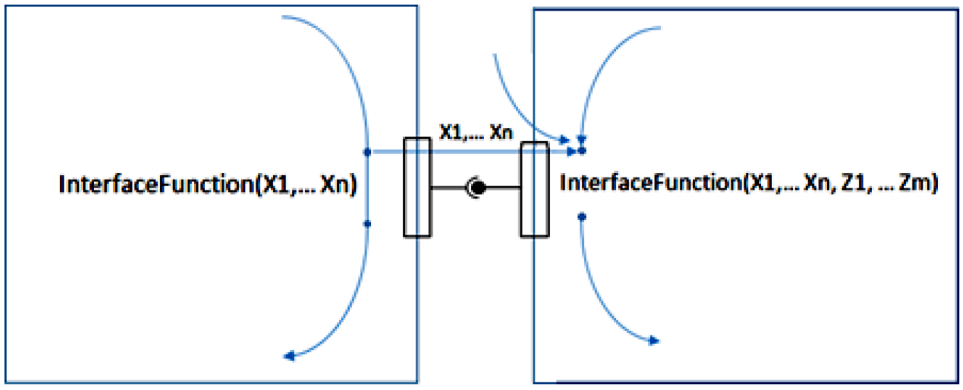

Fig. 14. DE, TT and TDF MoCs.

Table 1
Attributes defining MoC groups.

Group of MoCs	RtService exeKind	syncKind	Other Properties notAttendedService
FC/RPC/RMI	rem.Im.	sync/async	Any (except dynamic and other)
RV/CSP	rem.Im.	rendezvous	Any (except dynamic and other)
DF	Deferred	async /rendezvous	Any (except dynamic and other)
DE/TT/DTF	Deferred	async/ rendezvous	other
SR	rem.Im.	async	dynamic

Outputs will be generated by the component receiving the data and sent, in a similar way, to another component or to the external environment. DF communication is asynchronous. Components may generate data and consume data at any time. This means that in the general case, a buffer is needed to store data when, during some period, there are more date produced than consumed.

When the buffers between components never gets full (infinite capacity), DF becomes a Kahn Process Network (KPN). In real systems, buffers will have a finite size meaning that at certain points in time they may get full. In order to keep the properties of a KPN, the calling thread should stop. This may lead, eventually, to deadlocks.

When the interface function is called, the call is stored in the buffer to be attended afterwards. In that case, the execution of the calling thread

Table 2
RPC MoC alternatives.

Required Port NotAttendedService	retry	RtService concurrency	exeKind	syncKind	Provided Port queueSize	FullPoolPolicy	RPC semantic
infiniteWait	none	G or C	rem.Im.	sync.	none	none	exactly once
infiniteWait	none	G or C	rem.Im.	async.	none	none	at most once
timedWait	0	G or C	rem.Im.	sync.	none	none	exactly once
timedWait	0	G or C	rem.Im.	async.	none	none	at most once
timedWait	> 0	G or C	rem.Im.	sync.	none	none	at least once
timedWait	> 0	G or C	rem.Im.	async.	none	none	maybe once

Table 3
CSP and RV MoCs.

Required Port NotAttendedService	retry	RtService concurrency	exeKind	syncKind	Provided Port queueSize	FullPoolPolicy	MoC
infiniteWait	none	G or C	rem.Im.	rendezvous	none	none	CSP
timedWait	0	G or C	rem.Im.	rendezvous	none	none	RV
timedWait	> 0	G or C	rem.Im.	rendezvous	none	none	RV

time of the required function is large, the calling function sends the data to be processed and gets the data from the previous computation, which have been stored by the provider component after the previous call.

CSP may lead to deadlocks. In order to avoid them, a time-out can be defined. If 'retry' is set to '0', the calling function waits to be accepted during the timeout period. If it elapses, the function continues execution. If 'retry' is set to 'n', the function will be called at least 'n' times the timeout elapses. None of these cases corresponds to a CSP system. The following table shows the different alternatives:

5.4.3. Data flow (DF)

In a DF system, components communicate through data, which flow from the inputs of the system to internal components, among them and to the outputs. Thus, interface functions do not have output arguments.

continues. In order to ensure that the calling thread is able to continue, the interface function should not have any output parameter. If this happened, the calling function should wait until the function is finished and the output parameters calculated, thus, avoiding the call to be asynchronous. In the KPN MoC, the 'NotAttendedService' attribute is 'infiniteWait'. If the component behaves as an actor in which its internal behavior is executed each time a certain number of interface functions in its inputs have been called generating in each output a certain number of function calls, the MoC becomes Synchronous Data Flow (SDF). Depending on how many data are consumed (produced) in each input (output) each time, several variants of the fundamental SDF appear. If the number of data consumed (produced) in each input (output) is constant, the MoC is called multi-rate DF, regular DF or just SDF. A special case is when the rate in all the inputs and outputs is the same.

Table 4
DF MoC.

Required Port		RtService			Provided Port		MoC
NotAttendedService	retry	concurrency	exeKind	syncKind	queueSize	FullPoolPolicy	
infiniteWait	none	G or C	deferred	async.	> 0	block	KPN/SDF
infiniteWait	none	G or C	deferred	async.	> 0	(any other)	DF
dynamic	none	G or C	deferred	async.	> 0	any	DF
timedWait	0	G or C	deferred	async.	> 0	any	DF
timedWait	> 0	G or C	deferred	async.	> 0	any	DF

Table 5
DF MoC when data and computation are decoupled.

Required Port		RtService			Provided Port		MoC
NotAttendedService	retry	concurrency	exeKind	syncKind	queueSize	FullPoolPolicy	
infiniteWait	none	G	deferred	rendezvous	> 0	block	KPN/SDF
infiniteWait	none	G or C	deferred	rendezvous	> 0	(any other)	DF
dynamic	none	G or C	deferred	rendezvous	> 0	any	DF
timedWait	0	G or C	deferred	rendezvous	> 0	any	DF
timedWait	> 0	G or C	deferred	rendezvous	> 0	any	DF

The MoC in this case is called single-rate DF. If these rates change but following a static cyclic sequence of constant values, then the MoC is called cyclo-static DF. In all these cases, it is possible to find a static scheduling minimizing the required buffer sizes. This is not possible in the case of dynamic rates in dynamic Data Flow (DDF) systems [56].

The maximum number of function calls to be stored is defined with the attribute 'buffer'. It may happen that a higher production than consumption rates produces the buffer to get full. The policy to follow in that case is determined by the 'fullPoolPolicy' attribute. If the policy is 'block', and a new data is produced, the component is blocked until the buffer is read and, therefore, new free space is made available. The way around, if the thread in the provided component tries to read from an empty buffer, it is blocked until new data are produced and written. In both cases, a deadlock may be produced.

There are two ways to avoid deadlocks. The first one is to choose any other 'FullPoolPolicyKind' value. In this case, no new call is blocked but previous calls might be lost. The other possibility is to specify a timeout. In that case, if the timeout elapses, then the call is aborted if 'retry $= 0$' or a new call is tried if 'retry > 0'. None of these two ways corresponds to KPN or SDF models. The following table shows the different alternatives leading to KPN, SDF or simple DF, as shown in Table 4.

It is important to note that in KPN the order of the data is critical. Thus, concurrent operation is not accepted, in order to avoid order inversions.

A special case is found when the reading (writing) of the data is decoupled from the computation on the data. This happens when the interface function just gets the data to be read at any time by an internal thread in the component or puts the data to be read at any time by another component. Getting (putting) the data is so fast that there is no chance that a new getting (putting) of the data occurs when the previous data is still being got (put). In that case, if the goal is to ensure that no data is lost by a write-write or double-taken by a read-read, the RtService has to be implemented with a 'rendezvous' value for the 'syncKind' attribute, as shown in Table 5.

5.4.4. Discrete event (DE), Time-triggered (TT), Timed data flow (TDF)

Discrete Event, Time-Triggered and Timed Data Flow are MoCs where the functionality (internal to the component or a service call) is executed when triggered by an event. Thus, there are two elements to consider: the trigger and the functionality to be triggered.

- In the TT [57] MoC, the moment in which each component reads the inputs and the time in which it delivers the outputs are determined in advance.
- The TDF [47] MoC is basically a TT MoC in which the frequency of each component may be different depending on the input and output rates. This is the MoC used by many analog simulators like Modelica, Simulink and SystemC-AMS. In practice, the functionality first reads all data required to execute from the inputs, and then, it waits to the trigger to execute.
- In DE [58] systems, components react to events in their inputs created to control the execution of the services. An event is an instantaneous indication to trigger a reaction. Theoretically, it can be considered similar to DTF with delta times. In practice, the event is generated with all the services of the previous iteration have finished their execution. The automatically generated code must be in charge of that.

To model S3D, TDF and DE systems, the functionality waiting to be triggered will be stopped in a synchronization point, similar to the used in rendezvous to guarantee that new data has arrived. That way, when the trigger arrives the function is resumed. The trigger is generated by the glue code synthesized automatically in S3D.

The following is the representation of the interfaces in DE, TT and TDF:

These systems require delivering data to the next iteration, and a storage queue to execute correctly independently of the order of execution of the client and the server. Additionally, in TT and TDF the order of the data is important, and so, concurrent operation is not accepted in order to avoid order inversions. The result is shown in Table 6.

5.4.5. Synchronous reactive (SR)

In a SR system, the activity in the inputs, in our case, the calls for

Table 6
Attributes of TT, TDF and DE MoCs.

Required Port		RtService			Provided Port		MoC
NotAttendedService	retry	concurrency	exeKind	syncKind	queueSize	FullPoolPolicy	
other	none	G	deferred	async	>0	block	TT
other	none	G	deferred	rendezv.	>0	block	TDF / DE

Table 7
Attributes of SR MoC.

Required Port		RtService			Provided Port		MoC
NotAttendedService	retry	concurrency	exeKind	syncKind	queueSize	FullPoolPolicy	
dynamic	none	G or C	rem.Im.	async.	none	none	SR

required functions, trigger the internal activity among components until the system reaches a stable state in which no further function calls are made. This time, which in reality will be finite, is considered zero and all the activities performed are considered synchronous each other. Only then, new activities in the inputs are allowed. From this point of view, this model of computation does not impose any restriction to the properties in components and interfaces (Table 7).

In SR it is required to ensure that the new set of events starts when the system is stable. To do so, a global control component can be used, similarly to DE system.

6. Application example and simulation results

The modeling methodology has been applied to an avionics application provided by Thales, a safety-critical Flight Management System (FMS) in an airplane. The purpose of the FMS in modern avionics is to provide the crew with centralized control for the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management, and geographical situation information. From preset flight plans (take-off airport to landing airport), the FMS is responsible for the plane localization, the trajectory computation allowing the plane to follow the flight plan, and the reaction to pilot directives.

The FMS is decomposed into several tasks modeled as components. It computes various data (i.e., exact location, trajectories, and nearest airports list among many others) and it is supposed to send guidance instructions to the autopilot and also to send the different results to a display. The FMS inputs consist of:

- Sensor data: The FMS receives data from different sensors installed in a plane like: the GPS, inertia sensors, etc. The data from these sensors and state data conserved in the FMS is used to estimate an accurate position (including latitude, longitude and altitude) of the plane.
- Pilot commands: The pilot can configure most of the tasks in the FMS. For that purpose, the FMS is able to receive configuration commands.
- Navigation database: The FMS includes a read-only Navigation Database that contains the elements from which the flight plans are constructed, such as: airport locations, runway locations, departure procedures, airways (similar to sky-level highways), arrival procedures.

Likewise, the FMS outputs consist of:

- The display: All the data computed by the FMS could be displayed on a console or through lights/LEDs on panels. Each FMS task can send the data independently of the other tasks to the display.
- The autopilot: The role of the autopilot is to translate the attitude output information from the FMS (delta to be applied to the airplane roll, pitch and yew angles) to actions to be issued by the aircraft actuators like aps, slats, spoilers and the rudder.

The use case includes all the features of the kind of multi-domain applications the proposed methodology is targeting: data received from sensors, user commands, critical and non-critical computations and data outputs to be provided to the user with different criticalities and rates. The IoT link is represented by the access to the data base in the cloud.

The system has been modeled in S3D using Eclipse Neon EMT. The package of the components library is 174KB. The action language used to describe the functionality is C++. The corresponding code including a test bench with a complete flight from Santander to Paris is 42.4MB. Using the components in the library, the system is modeled in 1MB.

Table 8
Simulation results under the RPC MoC.

cores	MHz frequency	% use CPU1	CPU2	CPU3	CPU4	sec sim.time	events
1	1,000	26.7				4,022	392,074
	500	53.5				4,022	392,063
	250	99.6				4,022	364,679
	100	100				4,022	151,541
2	1,000	21.5	5.2			4,022	392,074
	500	38.5	14.9			4,022	392,074
	250	73.9	33.1			4,022	392,074
	100	99.9	91.7			4,022	289,612
4	1,000	21.4	3.8	1.4	0.1	4,022	392,074
	500	38.6	12.6	1.9	0.4	4,022	392,063
	250	73.5	29.5	3	1	4,022	392,086
	100	99.9	75.1	14.6	0	4,022	287,250

The S3D framework shows its reusability as the components were developed and stored in a library from where they have been taken to compose the FMS. Each generic component has its own entity ready to be used in other projects even with a different composition of the required and/or provided interfaces. The hierarchical partitioning capability supported by S3D both at the functional and the HW/SW platform levels shows its scalability. In order to show the S3D simplicity in exploring the design space, an RPC model of the system has been analyzed and its performance estimated under three architectural solutions with 1, 2 and 4 cores working at 4 different frequencies. It is worth mentioning that the 12 models were generated automatically from the PSMs resulting after changing just two numbers in the model, the number of cores in the node and their frequency. Under the RPC MoC, the main method in the 'RtUnit' components are periodic. Results are given in Table 8.

The resulting simulated time (sim.time) is the time taken by the airplane from the origin to destination. Just to give an idea of the complexity of the model, during this time, the airplane sensors generate values each 200 μsec, which is much higher than the typical period of the system task reading this information. The reason is double: on the one hand we want to generate a completely exact value depending on the jitter of the reading task; on the other hand it is a required for our environment model, which use linear equations to model a non-linear trajectory.

In the mono-core solution, the % of CPU usage grows as the core frequency decreases, as expected. At 250MHz CPU utilization is near 100%. Lower frequencies saturate the CPU. This affects the number of events, that is, the total number of times all the interface functions are called. As can be seen, this number decrease when the % of utilization grows to 90%, meaning that not all the required computations are made, and some data are lost. This effect is reduced with a 2-core solution. Now, the system can work correctly at 250MHz. Nevertheless, the behavior is again incorrect when the frequency is reduced to 100MHz. The performance of the system is very similar with 4 cores which means that adding two extra cores does not provide any advantage. This is because the utilization of CPU1 is not affected by a larger number of cores. A different scheduling policy might change this behavior leading to a more balanced solution.

In order to explore the flexibility of the methodology in supporting different MoCCs among components, two additional experiments are

Table 9
Simulation results under the RV MoC.

cores	MHz frequency	% use CPU1	CPU2	CPU3	CPU4	sec sim.time	events
1	1,000	58.5				4,022	704,831
	500	100				4,022	595,519
	250	100				7,785	567,371
	100	100				18,827	568,587
2	1,000	41	17.4			4,022	704,831
	500	82.2	34.8			4,022	704,831
	250	99.9	45.6			5,786	619,867
	100	99.9	50.2			14,013	644,129
4	1,000	41	11	0	6.5	4,022	704,832
	500	82.2	26.8	0	7.9	4,022	704,832
	250	99.9	40	4.5	2.33	5,999	650,594
	100	99.9	47.4	2.8	0.035	14,010	644,125

Table 10
Simulation results under the FIFO MoC.

cores	MHz frequency	% use CPU1	CPU2	CPU3	CPU4	sec sim.time	events
1	1,000	58.5				4,022	704,831
	500	99.9				4,022	564,666
	250	100				6,856	500,091
	100	100				19,243	591,969
2	1,000	41	17.4			4,022	704,831
	500	82.2	34.8			4,022	704,831
	250	99.8	54.2			4,022	456,342
	100	99.9	99.8			4,131	260,780
4	1,000	41	11	0	6.5	4,022	704,832
	500	82.2	26.8	0	7.9	4,022	704,832
	250	99.7	43.8	7.6	4.9	4,022	467,874
	100	99.9	77.9	18.7	5.1	4,022	255,758

made. In both cases, the intention is to reduce the number of events lost in order to increase the accuracy of the decisions taken by the FMS. In the first experiment, the MoC of some critical interface methods are changed to 'rendezvous' (RV). The main functions of the corresponding components are no longer periodic as they will be blocked in the RV points. Results are given in Table 9.

As it can be seen, the goal of processing more events is achieved, increasing from 392,074 to 704,831. Nevertheless, in the mono-core solution, at 250MHz and 100MHz, the simulated time increases. The explanation for this effect is the following. In the RPC MoC, the system reads less data from the environment, since reading task is not capable to execute on its period. As a consequence, the activity in the system decreases. This is demonstrated by the lower % of CPU utilization under the RPC MoC. If the synchronization mechanism between the system and the environment is a RV, the environment must wait until the generated data is taken. At a 1GHz and 500MHz, the system is fast enough to get all the data generated by the environment and compute it through all the system tasks. This does not happen at lower frequencies. The system cannot execute tasks on their expected periods, and therefore, some data are computed latter than expected. As the environment is Cyber-Physical, its behavior is no longer correct as the mechanical equations do not take into account that the time at which the data is computed is larger.

Increasing the number of cores does not solve the problem as more CPUs do not reduce significantly the % of CPU1 utilization, thus keeping the problem unchanged.

In a second experiment, some of the interfaces developed as RV, are implemented with FIFOs. Being an asynchronous MoC, some of the problems appearing with the RV should not happened. Again, main functions in components communicating via FIFOs are not periodic as they should read as much data as possible from the input FIFOs and produce as much data as possible in the output FIFOs, getting blocked only when an input FIFO is empty or an output FIFO is full. Results are given in Table 10.

As it can be seen, the asynchronous character of the FIFO removes some of the blocks seen before. Now, only the slowest of the 2-core solution and all the 4-cores solutions work without problems.

6.1. *Comparative analysis with the closest modeling methodologies*

The following table makes a comparative analysis among S3D, Chess and Hepsycode, the three closest UML/MARTE system modeling methodologies:

If we had use Chess, the modeling effort would have been very similar. In both tools, the UML model can be used for schedulability analysis using MAST [53]. Chess does not support the generation of a simulation model. From the MARTE model, an ADA template can be generated to be filled with the application code. Chess does not specify the MoCC among components so that it is not possible to explore different possibilities. As the ports and interfaces are fixed, reusability of the model is limited. Once the functional code for the component is developed, the MoCC of the component is fixed and any change would require rewriting the code. Thus, reusability is more difficult.

In the case of Hepsycode, the only MoCC supported is CPS. Hepsycode can generate a SystemC simulation model as the system components are described in this language. Application code generation would require a generation process from the SystemC model.

7. Conclusion

Model-Driven Engineering is a powerful mean to address the increasing complexity of real-time and embedded systems. The services that are starting to be developed to date and will become pervasive in the short term require scalable system modeling and design methodologies. In this paper, improvements to current practices in scalability and flexibility are proposed. They have been assessed on a complex use-case, a flight management system showing how a single-source approach can model the complete system with an unambiguous semantics to be used by the different design tools. The FMS project makes use of a library of previously modeled components, thus facilitating its reusability. Components in the library are generic and, therefore, easy to be adapted to different projects. Both the platform-independent and the platform description models are hierarchical. An essential aspect of flexibility improvement comes from the possibility to change the way in which component interact among them just by defining the properties in the interface methods and the required and provided ports leading to different MoCCs.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Table 11
CSP and RV MoCs.

Modeling Framework	Library-based	MoCC flexibility	Schedulability Analysis	Executable Model Generation	Application Code Generation
S3D	Yes	Yes	Yes(with MAST)	Yes(Native simulation)	Yes
Chess	No	NoUp to the user	Yes(with MAST)	No	Partially(ADA template)
Hepsycode	No	No(Only CSP)	No	Yes(SystemC)	No

Acknowledgments

This work has been partially funded by the EU and the Spanish MICINN through the ECSEL MegaMart and Comp4Drones projects and the TEC2017-86722-C4-3-R PLATINO project.

References

[1] L. Jozwiak, "Advanced mobile and wearable systems, in: Microprocessors and Microsystems, V.50, Elsevier, 2017, pp. 202–221.
[2] H. Posadas, S. Real, E. Villar, "M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration, in: C. Silvano, W. Fornaciari, E. Villar (Eds.), Multi-objective Design Space Exploration of Multiprocessor SoC Architectures: the MULTICUBE Approach, Springer, 2011.
[3] M. Brambilla, J. Cabot, M. Wimmer, "Model-Driven Software Engineering in Practice, Morgan & Claypool (2012).
[4] S.Ambler: "Single source information: an Agile best practice for effective documentation", 2015, http://agilemodeling.com/essays/singleSourceInformation.htm.
[5] E. di Nitto, Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach, in: P. Matthews, D. Petcu, A. Solberg (Eds.), Springer Open, 2017.
[6] B. Selic, S. Gerard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems, Morgan-Kaufman, 2014.
[7] S. Korra, S.V. Raju, A.V. Babu, "Strategies for Designing and Building Reusable Software Components, Int. J. Comp. Sci. Inform. Tech. 4 (N.5) (2013). V.
[8] A. Kleppe, J. Warmer, W. Blast, MDA explained: The Model-Driven Architecture: Practice and Promise, Addison-Wesley, 2004.
[9] B. Tekinerdogan, M. Aksit, F. Henninger, "Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach, Electron. Note Theor. Comp. Sci. 163 (2) (2007) 45–64. VI.
[10] "S3D: Single-Source System Modeling", s3d.unican.es.
[11] J.E. Rumbaugh, M. Blaha, W.J. Premerlani, F. Eddy, W.E. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1991.
[12] E. Lee, "The Problem with Threads, Computer 39 (5) (2006) 33–42. VN.
[13] E. Lee, "Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design, in: Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, 2003 citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.230.1295&rep=rep1&type=pdf.
[14] N.R. Jennings, On agent-based software engineering. Artificial Intelligence, Elsevier, 2000, p. 117.
[15] E. Lee, S. Neuendorffer, "Actor-Oriented Models for Codesign, in: R. Gupta, P. L. Guernic, S.K. Shukla, JP. Talpin (Eds.), Formal Methods and Models for System Design, Springer, 2004.
[16] D.K. Chaturvedi, Modeling and Simulation of Systems Using MATLAB and Simulink, CRC Press, 2010.
[17] P Srinivas, P.D.P. Rao, K.V. Lakshmi, "Modelling and Simulation of Complex Control Systems using LabView, Int. J. Contr. Theo. Comp. Model. V.2 (4) (July, 2012). N.
[18] M. Otter, H. Elmqvist, M. Hilding, S.E. Mattsson, "Multi-Domain Modeling with Modelica, in: P.A. Fishwick (Ed.), Handbook of dynamic system modeling, CRC Press, 2007.
[19] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "Taming heterogeneity-the Ptolemy approach, in: Proceedings of the IEEE V.91, IEEE, 2003.
[20] B. Spitznagel, D. Garlan, "A compositional approach for constructing connectors, in: proc. of the Working IEEE/IFIP Conference on Software Architecture, IEEE, 2001, pp. 148–157.
[21] https://es.mathworks.com/solutions/internet-of-things.html.
[22] https://www.intel.es/content/www/es/es/cofluent/cofluent-technology-for-iot.html.
[23] D. Bálek, F. Plášil, "Software Connectors and Their Role in Component Deployment, in: K. Zieliński, K. Geihs, A. Laurentowski (Eds.), New Developments in Distributed Applications and Interoperable Systems, 70, Springer, 2001. V.
[24] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, "Abstractions for software architecture and tools to support them, IEEE Trans. Softw. Eng. 21 (4) (1995). IV.
[25] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Pearson Education, 2010.
[26] N.R. Mehta, N. Medvidovic, S. Phadke, "Towardsa taxonomy of software connectors, in: proc. of the 22nd International Conference on Software Engineering (ICSE '00), ACM, 2000, pp. 178–187.
[27] J. Cámara, D. Garlan, B. Schmerl, "Synthesis and Quantitative Verification of Tradeoff Spaces for Families of Software Systems, in: A. Lopes, R. de Lemos (Eds.), Software Architecture: 11th European Conference, ECSA, 2017. Canterbury, UK, September 11-15, 2017, Proceedings", Springer, 2017.
[28] "Semantics of a Foundational Subset for Executable UML Models", OMG, 2018, https://www.omg.org/spec/FUML/About-FUML/.
[29] http://docs.metamorphsoftware.com/doc/getting_started/introduction/introduction.html.
[30] F. Ciccozzi, I. Crnkovic, D. di Ruscio, I. Malavolta, P. Pelliccione, R. Spalazzese, "Model-Driven Engineering for Mission-Critical IoT Systems, IEEE Softw. V.34 (1) (Jan.-Feb. 2017). I.
[31] P.H. Feiler, D.P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley, 2012.
[32] D. Delanote, S. Van Baelen, W. Joosen, Yolande Berbers, "Using AADL in model driven development, in: proc. IEEE International Conference on Engineering Complex Computer Systems (ICECCS), 2007.
[33] https://www.modelio.org/.
[34] https://www.eclipse.org/papyrus/.
[35] F. Herrera, J. Medina, E. Villar, "Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design approach, in: Soonhoi Ha, Jürgen Teich (Eds.), Handbook of Hardware/Software Codesign, Springer, 2017.
[36] https://www.polarsys.org/time4sys/.
[37] A. Gamatié, S. Le Beux, É. Piel, R.B. Atitallah, A. Etien, P. Marquet, J-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems, ACM Trans. Embed. Comput. Syst. (TECS) 10 (4) (2011) 1–36. VN.
[38] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio A. Zovi, T. Vardanega, "CHESS: A model-driven engineering tool environment for aiding the development of complex industrial systems, in: proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2012.
[39] M. Panunzio, T. Vardanega, "A component-based process with separation of concerns for the development of embedded real-time software systems, J. Syst. Softw. V.96 (2014) 105–121.
[40] L. Baracchi, A. Cimatti, G. Garcia, S. Mazzini, S. Puri, S. Tonetta, "Requirements Refinement and Component Reuse: The FoReVer Contract-Based Approach, in: A. Bagnato, L.S. Indrusiak, I.R. Quadri, M. Rossi (Eds.), Handbook of Research on Embedded Systems Design, IGI-Global, 2014.
[41] V. Muttillo, G. Valente, Luigi Pomante, "Criticality-aware Design Space Exploration for Mixed-Criticality Embedded Systems, in: proc. Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE), 2018.
[42] K.A. Weiss, E.C. Ong, N.G. Leveson, "Reusable Specification Components for Model-Driven Development, in: proc. of the International Conference on System Engineering, INCOSE, 2003.
[43] U. Shani, H. Broodney, "Reuse in model-based systems engineering, in: proc. of the 2015 Annual IEEE Systems Conference, SysCon, 2015, pp. 77–83.
[44] S. Korra, D. Raju, A.V. Babu, "Strategies forDesigning and Building Reusable Software Components, Int. J. Comp. Sci. Inform. Tech. 4 (5) (2013) 655–659. N.
[45] P. Bogdan, R. Marculescu, "Cyberphysical Systems: Workload Modeling and Design Optimization, IEEE Des. Test Comp. 28 (4) (July-Aug, 2011). VN.
[46] P. Bogdan, "A cyber-physical systems approach to personalized medicine: Challenges and opportunities for NoC-based multicore platforms, in: proc. of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, 2015.
[47] M. Barnasconi, K. Einwich, C. Grimm, T. Maehne, A. Vachoux, Advancing the SystemC Analog/Mixed-Signal (AMS) Extensions: Introducing Dynamic Timed Data Flow, Open Syst. C Initiat. (2011). www.accellera.org/images/resources/articles/amsdynamictdf/Whitepaper_SystemC_AMS_Dynamic_TDF_September_2011.pdf.
[48] Y. Xiao, Y. Xue, S. Nazarian, P. Bogdan, "A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach, in: proc. of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, 2017.
[49] Y. Xiao, S. Nazarian, P. Bogdan, "Self-Optimizing and Self-Programming Computing Systems: A Combined Compiler, Complex Networks, and Machine Learning Approach, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 27 (6) (June 2019). VN.
[50] R. Giorgi, "Exploring dataflow-based thread level parallelism in cyber-physical systems, in: proc. of the ACM International Conference on Computing Frontiers (CF '16), New York, ACM, 2016.
[51] essyn.unican.es.
[52] vippe.unican.es.
[53] M. Harbour, J.J. Gutierrez, J. Palencia, J.M.D. Moyano, "MAST: Modeling and analysis suite for real time applications, in: proc. Euromicro Conference on Real-Time Systems, 2001.
[54] K-K. Lau, S. di Cola, An Introduction to Component-Based Software Development, World Scientific Publishing Company, 2017.
[55] Network Working Group, RPC: Remote Procedure Call Protocol Specification Version 2, Sun Microsystems, 2009. https://tools.ietf.org/html/rfc5531.
[56] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, "Cyclo-static data flow, in: proc. of the International Conference on Acoustics, Speech, and Signal Processing, IEEE, 1996.
[57] H. Kopetz, The time-triggered model of computation, in: Proceedings 19th IEEE Real-Time Systems Symposium, 1998.
[58] L. Thiele, Discrete Event Systems - Introduction, Distrib. Comput. ETH Zurich (2019). https://pdfs.semanticscholar.org/5a22/af628426a44c0bcbddd26b4c31c44a99af35.pdf.