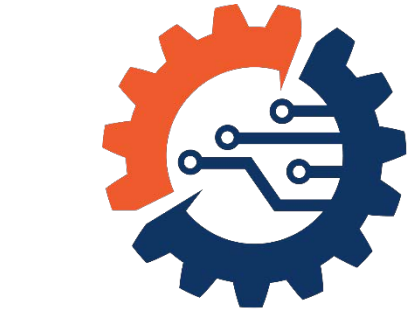


The S3D Modeling Methodology v1.0

February 2020
Javier Merino
Eugenio Villar
Hector Posadas

Document history

6-FEBRUARY-2019	FIRST DRAFT BY E. VILLAR AND H. POSADAS
15-OCTOBER-2019	DRAFT VERSION V.02 BY J. MERINO
7-FEBRUARY-2020	REVISION BY E. VILLAR

Executive summary

In this document, the S3D modeling methodology is detailed. The methodology is built up on the previous methodology refined in the Pharaon and Contrex projects. It is derived from the requirements imposed by the evolution of embedded systems from connected but isolated boards in a product to a component of a network of distributed devices connected among them and with the cloud in order to provide a certain value-added service to the final users. This radical new context requires the capability to model complex, heterogeneous, distributed systems while ensuring scalability and reusability.

To achieve these goals new modeling methods have been developed. In order to improve scalability, hierarchical partition of both the application functionality and the executive HW/SW platform have been better supported. In order to improve reusability, the concept of generic component has been defined. In addition, these generic components are grouped in libraries so that they can be reused as many times as required.

The new S3D modeling methodology requires an adaptation of the associated tools, VIPPE for system simulation and performance analysis and eSSYN for SW synthesis. These new versions of the tools are being assessed on the Thales FMS Use Case. Plans exist to apply them in other use cases, such as the Nokia Base Transceiver Station.

Table of Contents

Acronyms

AL	Action Language used to specify the functionality
ASHW	Application-Specific Hardware
ASIC	Application-Specific Integrated Circuit
CPSoS	Cyber-Physical Systems of Systems
CPU	Central Processing Unit
DSP	Digital Signal Processor
GPU	Graphics Processing Unit
HdS	Hardware-dependent Software
IP-XACT	IEEE standard for IP Packaging, Integration and Reuse
ISA	Instruction Set Architecture
MDA	Model-Driven Architecture
MoC	Model of Computation
PIM	Platform-Independent Model
PDM	Platform Description Model
PSM	Platform-Specific Model
S3D	Single-Source System Design framework developed by the University of Cantabria
SoS	System-of-Systems
UC	University of Cantabria

1 Introduction

Model-Driven Software Engineering has proven to be a powerful approach to deal with the increasing complexity of software development [BCW12]. It can be adapted to different design contexts and domains, being compatible with methodologies like Agile [Amb15] and DevOps [NMP17]. Currently, most systems involve just a small number of computing resources, such as a datacenter processing the voice from a smartphone and providing a voice-to-text service, or the distance sensors in a car connected to an Electronic Control Unit providing an automatic parking service to the driver. In these examples, specifying the complete service, deciding which functionality to execute in each node, and programming the corresponding application, although reasonably complex, are affordable tasks. However, services in a fully interconnected world will be composed of many SW components deployed on multiple devices of many kinds. All these electronic devices and the distributed SW they execute compose a system of a high complexity in terms of the distributed executive platform, the functionality it implements and the strong interaction with the physical environment that the system realizes. An additional, important aspect to consider is the interaction between the system and the humans both as users or involved directly or indirectly in its operation (humans in the loop). Is in this heterogeneous, multi-domain environment where the abstraction, inter-operability and reusability capabilities of Model-Driven Engineering become especially relevant. In this new context, Software Engineering is still an important part of the problem but no longer the only one. Understanding the underlying infrastructure of hardware devices and networks on which the functionality is deployed as well as its interaction with the physical world and the human being are of paramount importance.

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) [BCW12]. Nevertheless, the evolution commented above requires of new, holistic, mega-modeling methodologies able to model the complete system and its interaction with the physical environment in a unified way, thus supporting the verification of the functional requirements and the analysis of the non-functional requirements such as execution times, delays, data movement, power consumption, etc. Among these DSLs, UML/MARTE has been proposed for the modeling and analysis of real-time and embedded systems. The profile covers both system engineering by supporting the general resource and component modeling and software engineering, by supporting the high-level application modeling and the detailed software resource modeling. 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 [SeGe14].

The Microelectronics Group of the University of Cantabria (UC) has a large experience in system modeling using UML/MARTE. The modeling methodology has been improved along the time [PHV10] [MMW11][HPP12][GHH13][HPP14][PNP14][HMV17] extending its modeling capabilities. The final goal is to support efficient modeling of services implemented as Cyber-Physical Systems of Systems (CPSoS) [MVH17]. In this document, the S3D system modeling methodology based on UML/MARTE is proposed able to support efficient SoS modeling. In the next section the proposed system design methodology is described. The main improvements to current common practices are highlighted. S3D is a Single-Source, System Design Framework where 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 single-source 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) and Aspect-Oriented Software Development (AOSD) [Lap07] [TAH07]. From the central repository, different tools can be used in order to perform the different design tasks such as verification, simulation, performance analysis, schedulablility analysis, etc. Finally, when the design is considered correct, satisfying all the functional and extra-functional constraints, the code to be deployed on the different computational nodes of the distributed platform is automatically generated, as shown in Figure 1:

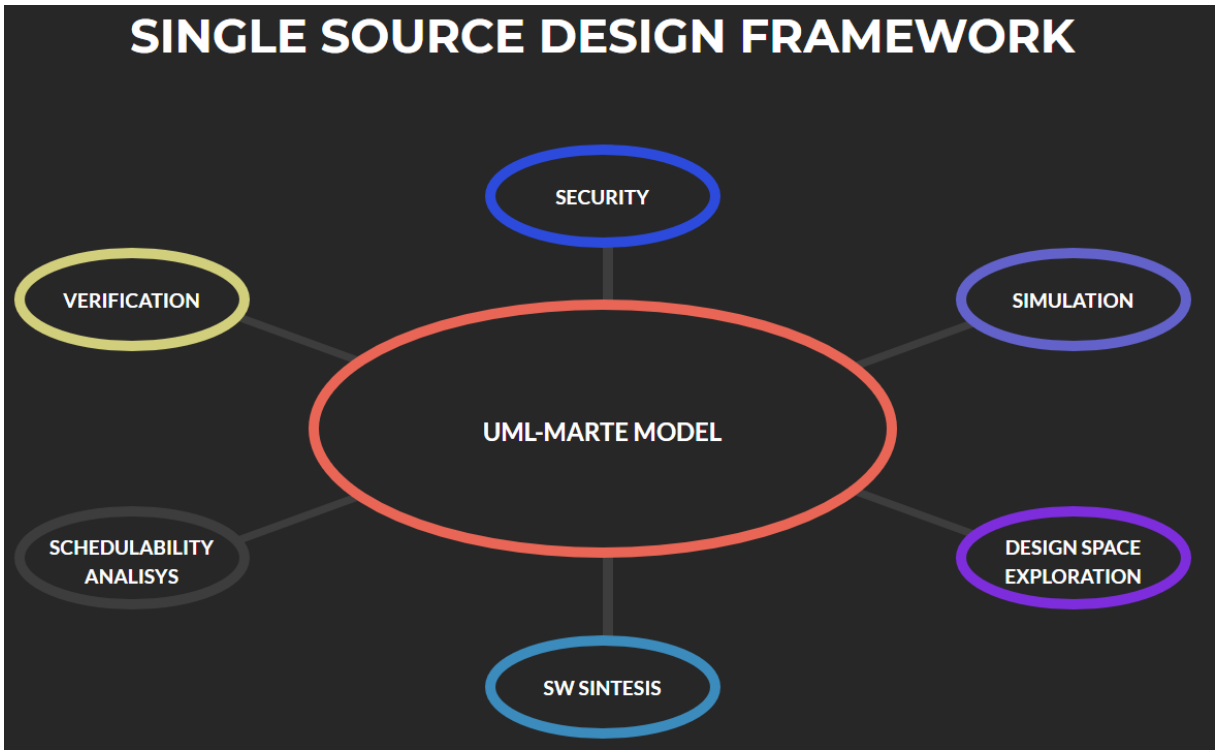


Figure 1 The S3D framework.

In this document, the S3D modeling methodology is described. The example used along the text is a Flight Management System (FSM) proposed by Thales as Use Case in the MegaMart project.

1.1 System design methodology

The modeling methodology proposed may be used in many different system design and verification methodologies. We will refer to the V-Model, to which the modeling methodology, as part of the S3D Framework, will be applied. The traditional V-Model is drawn in Figure 2. The descendent (left-side) steps correspond to design activities at system, architecture and component levels) while the ascendant (right-side) steps correspond to verification steps. Independently of the quality of the software testing methods used, software verification is usually limited to just functional verification, that is, the code is executed in the same machine it was developed under a collection of tests. In this way, design tasks such as design-space exploration, code optimization, architectural mapping, etc. in which performance metrics play an essential role in taking the right decisions, can be done only during the prototyping phase. In most cases, this is too late and any change would require a high re-designing effort. This problem is even harder in current heterogeneous architectures in which, even in a single platform, there

is a number of different computational devices (big-little CPUs, GPUs, DSPs, ASHW, etc.) on which to map the application components.

In the MegaMart project, the UC has the objective of reducing the development time for complex systems by exploring several improvements to the design and verification process. The first is the use of a multi-level verification framework introducing Model-in-the-Loop in addition to the Functional Verification commented above (SW-in-the-Loop) and the final prototyping (HW-in-the-Loop).

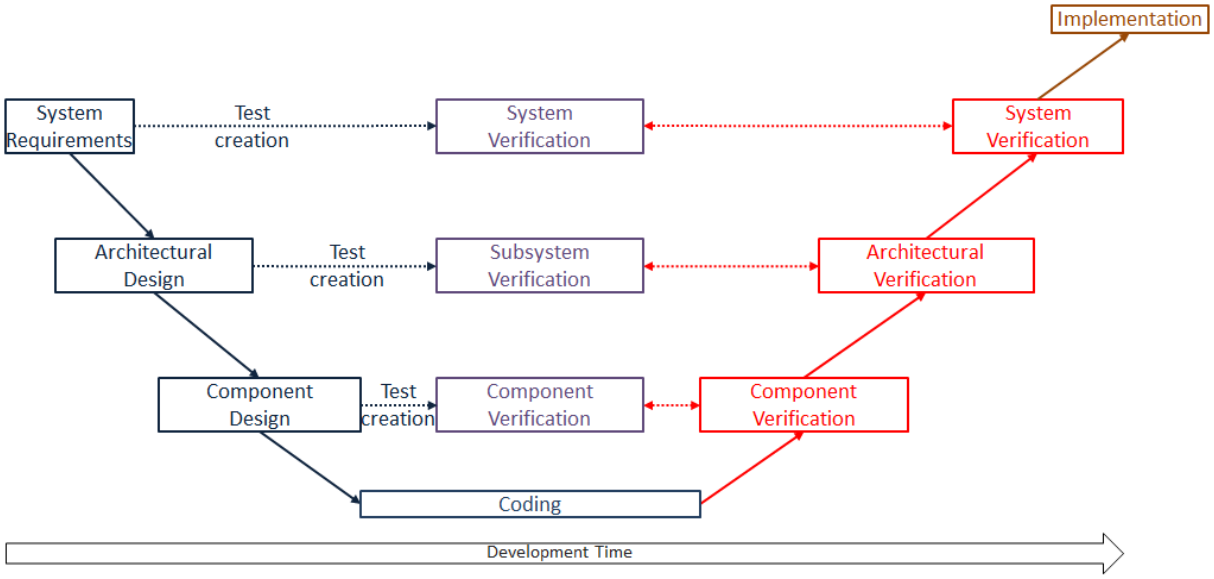


Figure 2 Traditional V-Model for SW Engineering.

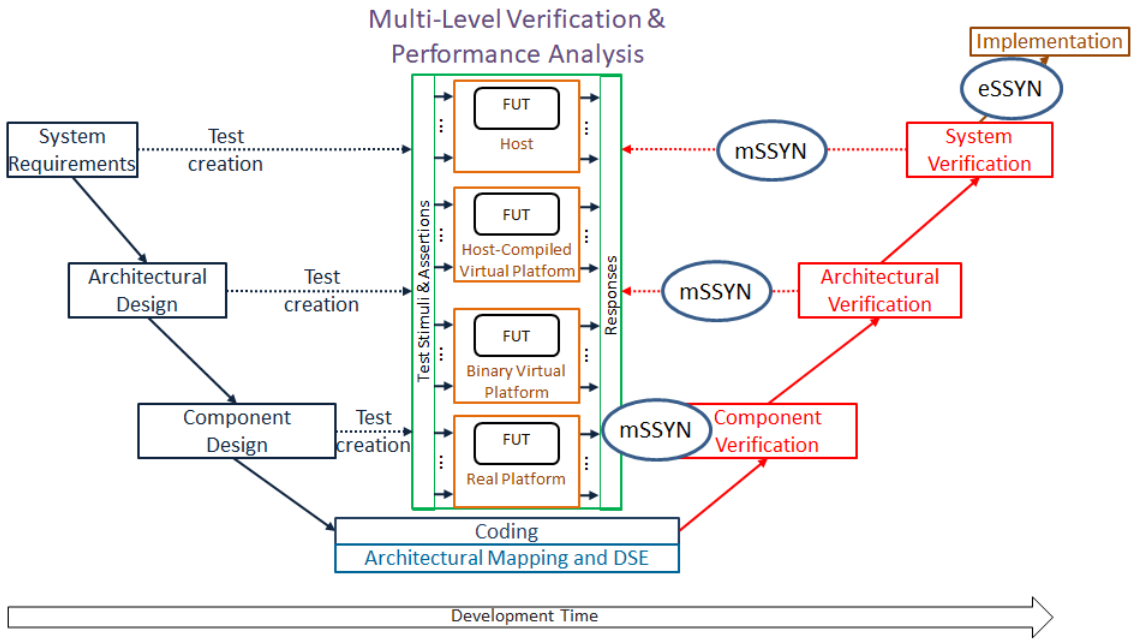


Figure 3 Some MegaMart improvements to the traditional V-Model.

In this way, both functional and extra-functional design mistakes can be detected earlier and corrected with much less effort and time. The second, by automatizing all the processes in the flow in which information is extracted from the Single-Source Model in order to perform a particular design task such

as executable model generation or final implementation. Both improvements are shown in Figure 3. As it can be seen, the multi-level validation & verification framework using simulation at different abstraction levels supports detection of design errors earlier in the design process. Reusability of the tests to be applied along the verification process is another key improvement. The models at each stage are automatically generated from the system model, thus accelerating the process. So, mSSYN generates the different models at different abstraction levels to be validated using several simulation technologies, when needed. At the end of the process, the final solution is automatically generated using eSSYN (essyn.com).

1.2 System modeling requirements

Nowadays, we have just started to realize the enormous potential of an interconnected world of billions of smart devices providing new services to people [Wel16]. The end of Moore's Law might facilitate the proliferation of new electronic systems supporting these new services. As commented above, these, services will be built of many SW components deployed on multiple devices, from small sensing motes, embedded systems and smartphones to data-centers, and even, High-Performance Computing (HPC) facilities. Despite its apparent diversity, all of them may be implemented with integrated systems containing heterogeneous processing elements (i.e. CPUs of different kinds, GPUs, DSPs and HW co-processors). In all cases, the systems will need to satisfy functional and extra-functional critical constraints, including safety, security, power efficiency, performance, size, and cost. The global characteristics of the system as a whole will depend on the characteristics of their independent components, but also on the interaction with the physical environment and among them through the different communication networks. Therefore, the main innovation in the time to come shall be to jump from the design of cyber-physical systems (CPS) to cyber-physical systems of systems (CPSoS). These complex, heterogeneous, distributed systems require an interdisciplinary approach where the knowledge about the physical side of the systems is indispensable to arrive at solutions that are taken up in the real world. To integrate these diverse research and development communities is the most crucial aspect for a successful future development of CPSoS. Current domain-specific methods are becoming obsolete; hence new predictive, engineering and programming methods and tools are required ensuring the satisfaction of the functional and extra-functional constraints imposed to the system while considering its interaction with the physical world and the humans.

The main reaction to the continuous evolution of computing platforms has been to decouple the application SW from the underlying HW. To achieve this goal many abstraction layers of middleware, communication protocols, operating systems, hypervisors and HW abstraction layers are being used. This approach is powerful enough for general-purpose systems, for which extra-functional constraints such as execution times, energy efficiency, dependability, etc. are not strict. But the technological evolution towards CPSoS based on heterogeneous devices composed of CPUs of different kind, GPUs, DSPs, HW co-processors etc., added to the need to satisfy stricter non-functional properties makes this goal unrealizable. Therefore, there is a need for a holistic modeling framework, across SW and HW layers, applications and domains. This modeling framework should be able to capture the complete high-abstraction model, integrating projects with different constraints (i.e. commercial or critical SW) and domains (i.e. from High Performance Computing, to embedded SW). The modeling and design framework should provide him/her with an accurate knowledge of the implications that the final implementation of the functionality on the concrete (distributed) platforms under a specific functional mapping will have in terms of extra-functional constraints. Beyond performance, energy consumption, safety, data traffic, security, adaptability, scalability, complexity management and cost-effectiveness have to be taken into account. This information about the complete system characteristics can be used in its design-space exploration and optimization. As commented above, an essential aspect to be taken

into account is the interaction of the system with humans all along the life cycle, since the specification and design of the system until its deployment, field and obsolescence.

UML has the potential to be the central modeling language in this new context. To achieve this goal, a consensus on a profile, powerful enough to capture all the relevant concepts required in CPSoS engineering while, at the same time, simple enough to find wide acceptance by the design community, is required. MARTE is a good starting point for two main reasons. Firstly, it captures most of the concepts required in system engineering on heterogeneous platforms under strict design constraints. Secondly, there is clear convergence among computing platforms and today, it is possible to find the same computing resources (i.e. CPUs, GPUs, and application-specific HW) in platforms apparently as different as an embedded system, a smartphone and a supercomputer.

Concrete requirements that the UML/MARTE modeling methodology should satisfy, follow.

1.2.1 Simplicity

Modeling is a time-consuming, error-prone activity. In order to minimize the modeling effort and to reduce the number of modeling mistakes, the modeling methodology should be simple, easy to understand and to be applied. The single-source modeling approach [Amb15] followed by S3D is intended to reduce the modeling effort. It supports capturing all the relevant information in a single model thus avoiding duplication of design information.

As an additional characteristic towards simplicity and understandability, the number of fundamental modeling primitives should be as reduced as possible. In our case, the methodology is Component-Based [LaCo17] and therefore, the fundamental modeling primitives are those shown in Figure 4

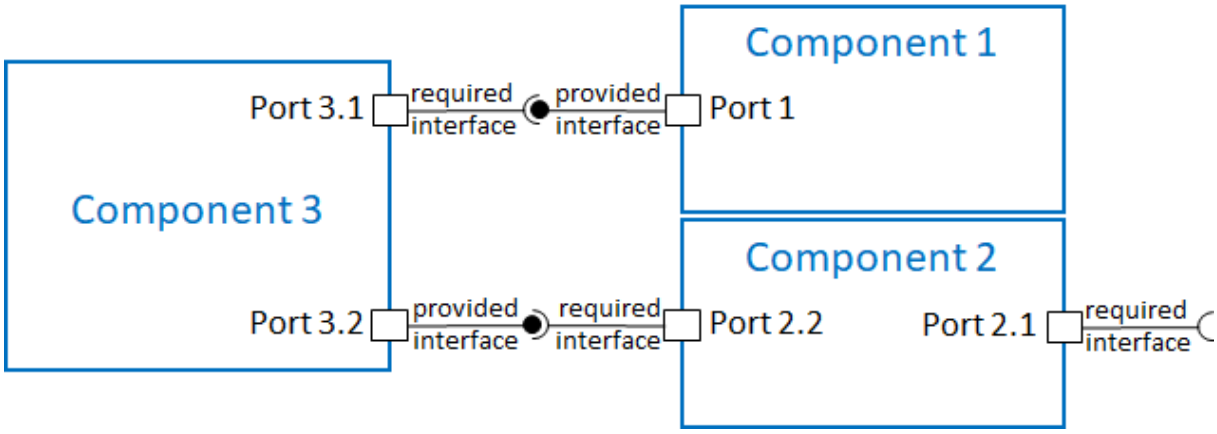


Figure 4 A system with three components.

The fundamental modeling element is the component. Components communicate among them through ports. The ports contain interfaces, which implement the communication methods. The components either require communication methods (or services) trough required interfaces or offer communication methods (or services) through provided interfaces. Just by looking to the system architecture in Figure 4, one can realize that ‘Component 2’ is an active component (stereotyped as an RTUnit) requiring services provided by other components. ‘Component 1’ on the contrary, is a passive component (which can be stereotyped as a PPUnit) providing services to other components. ‘Component 3‘ may have its own internal concurrent activity as it requires services through ‘Port 3.1’ and provides services through ‘Port 3.2’ or it is a PPUnit requiring services from a third component (‘Port 3.1’) in order to implement the services it provides through ‘Port 3.2’.

As it will be shown latter, this simple modeling mechanism will be able to support different system engineering methodologies and Models of Computation (MoC).

1.2.2 Scalability

Although simple, the modeling methodology should be able to support the modeling and design of complex systems. Systems providing the services commented above, implemented by the interaction of many different functional components deployed on many different computing devices of many kinds.

In order to achieve this goal, several fundamental techniques are supported. The first one, **hierarchy**. 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. In relation with hierarchy, another characteristic to be covered is **composability**. The components should be able to be composed 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 Figure 5.

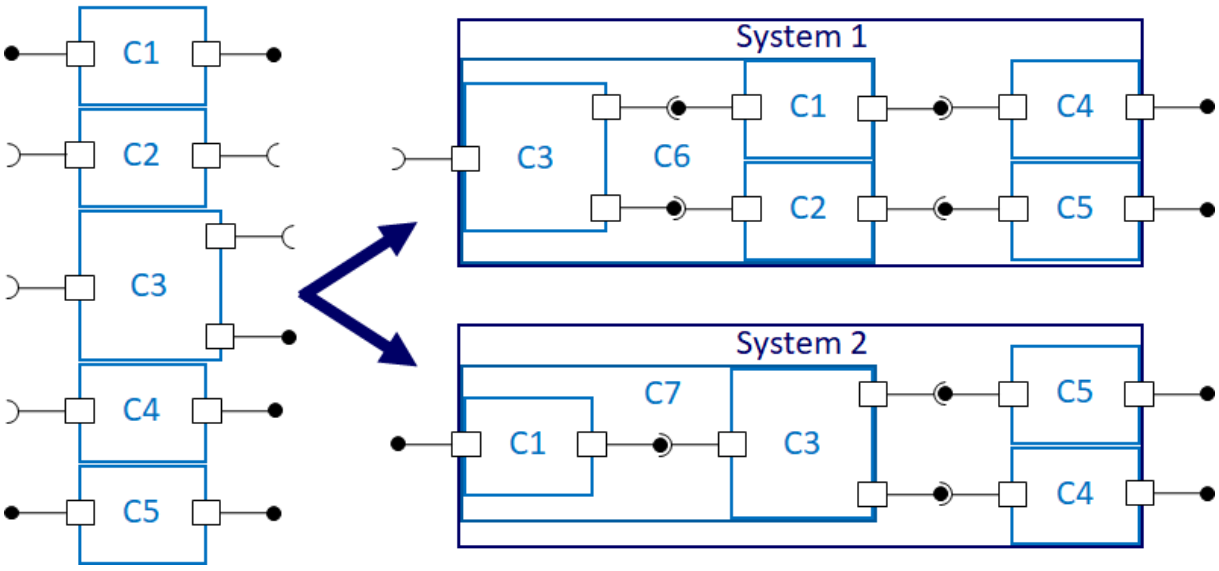


Figure 5 Different systems composed from the same components.

One of the main objectives of Megamart is the modeling of very complex systems requiring MegaModeling methodologies. This goal will be achieved by improving the modeling methodology proposed in this document with additional features like hierarchical HW modeling and multi-language support.

1.2.3 Separation of concerns

In the general case, capturing all the relevant information about the system in a single place following a single-source approach may be in contradiction with the simplicity goal stated above. In order to avoid this contradiction and to reduce the modeling effort, the system model is divided in 'views'. Each 'view' will capture all the relevant information about a specific design concern (i.e. the data types used, the functionality, the communication among components, the functional application, the system verification, etc.).

1.2.4 Design-Space exploration

The methodology should be flexible enough to support the analysis and comparison of many different architectural solutions for the 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. The concrete analysis model of a particular architectural mapping should be generated automatically [PRV11].

1.2.5 Reusability

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. The achievement of this goal requires the components completely platform independent. This provides important advantages in terms of reusability in two main aspects. The first one is the improvement in reusability of the components from project to project, that is, when there is a need to up-date the service. As each component encapsulates its functionality in a platform independent way, only those components whose functionality needs to be up-dated have to be considered as the services provided or required by the other components will not be affected even in case the up-date of the system requires a complete architectural re-mapping. This is particularly interesting in DevCons methodologies where the analysis of the behavior of the system in runtime allows the improvement of new versions of the same system or even, new generations of the product.

The second advantage comes from the reusability of the components when the execution platform is improved (i.e. new versions of the same family of platforms) or changed both in terms of the HW-dependent SW (i.e. a change in the OS or the middleware) or the HW. Apart from the reusability facilitated by the platform-independence of the components, there are other two aspects of S3D improving reusability. The first is the use of what we call 'Generic Components'. These components are defined only by the services they provide and/or require. They lack ports, as they will be added when the Generic component is instantiated as an 'Application Component' inside a concrete system, with the sole exception that a component requires the same interface from N providers, where it is needed that these N ports are specified in the component. The second is the use of interface inheritance. This allows the connection of components even if they provide/require different services whenever one interface inherits from the other.

Improving the reusability of a component requires an extra effort in encapsulating the component in a convenient way and integrating it in a library with related components. This effort is worth to be spent whenever the component is going to be optimized in new versions of the application or reused in a new application [KRB13].

Only once mapped to a concrete computing resource, the corresponding platform specific code including the required middleware, input-output access code and system calls should be generated. Our goal is to make this generation process completely automatic by SW synthesis.

2 System Modeling

In this section, the S3D modeling methodology will be detailed. First, the fundamental concepts supporting the methodology are described. Then, the views used in order to model the different aspects of the system ensuring a strong separation of concerns, are presented.

2.1 Fundamental elements

The system views are divided in two large groups, the Design & Verification (D&V) views and the Tool-Specific views. The former provides all the relevant information about the system in order to support its design, simulation, verification, performance analysis and synthesis. The latter include additional information required by specific tools supporting concrete design tasks. In this document, only the D&V views will be described.

The D&V views are divided in three groups, the Platform-Independent Model (PIM), the Platform Description Model (PDM) and the Platform-Specific Model (PSM).

2.1.1 Platform-Independent Model

In this section, the fundamental elements of the PIM will be described. The PIM captures all the information required to describe the platform-independent functionality of the system. Following the basics of Model-Driven Architecture (MDA), the PIM "exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms" [Tru06]. As the code is developed 'for' a particular platform, making use of platform-dependent code (system calls, drivers, etc.), the functionality of the objects in the model has to be expressed using abstract formats such as state-machines or sequence diagrams. From them it is possible to automatically generate the equivalent functional code. Nevertheless, experience shows that these means are useful only for small pieces of code but they fail when dealing with complex functions. Therefore, in S3D, as soon as the component is fully specified its functionality is developed by using the preferred programming language (i.e. C++, Java, etc.). Several codes can be associated to each component. In order to be Platform-Independent, the code should not include any system call or Hardware-dependent Software (HdS).

Generic components

As commented above, the system is conceived as a network of components. In order to maximize reusability and flexibility, components are generic components. In MARTE, these elements are *Structured Components*. In their most abstract form, the only external information about such elements is the services (functions) they provide and/or require, as shown in Figure 6. 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.

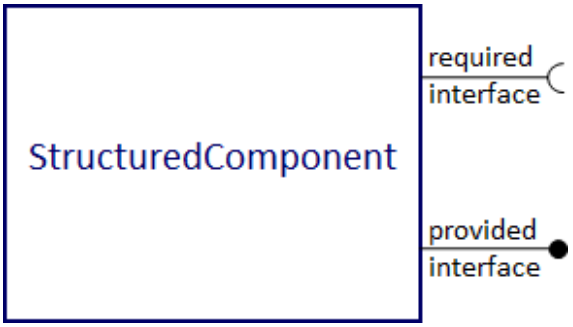


Figure 6 A Structured Component.

Each structured component will be linked to the file where its structured data and behavior is specified in the action language used, in our case, C++. In principal, no restrictions are imposed to the way the

behavior is specified. Nevertheless, as it will be described in more detail afterwards, component-based design methodologies impose restrictions on how the components interact among them, that is, only through well-defined interfaces [LaDi17]. Going further, actor-oriented design methodologies impose additional restrictions on the internal functionality of the component.

These Generic Components will be grouped and provided to the system engineer in libraries avoiding the need to develop them from scratch or from the adaptation of legacy code from previous projects.

Application components

From the generic components, application components with concrete ports and interfaces will be derived by inheritance, as shown in Figure 7. These application components can be instantiated as <<RtUnit>> or <<PpUnit>> weather they are an active, concurrent object or a passive one. The system will be obtained as a composition of such application components connected each other through concrete, compatible ports and interfaces. The behavior of the application component is the same as the generic component from which it inherits. As it will be explained later, in Section 0, the interaction among components can be specified in a flexible way by concrete properties. Depending on the properties assigned to the ports and interfaces in both sides, different MoCs can be supported. Functional and extra-functional constraints may be imposed to the application components using appropriate constraint-specification languages such as OCL.

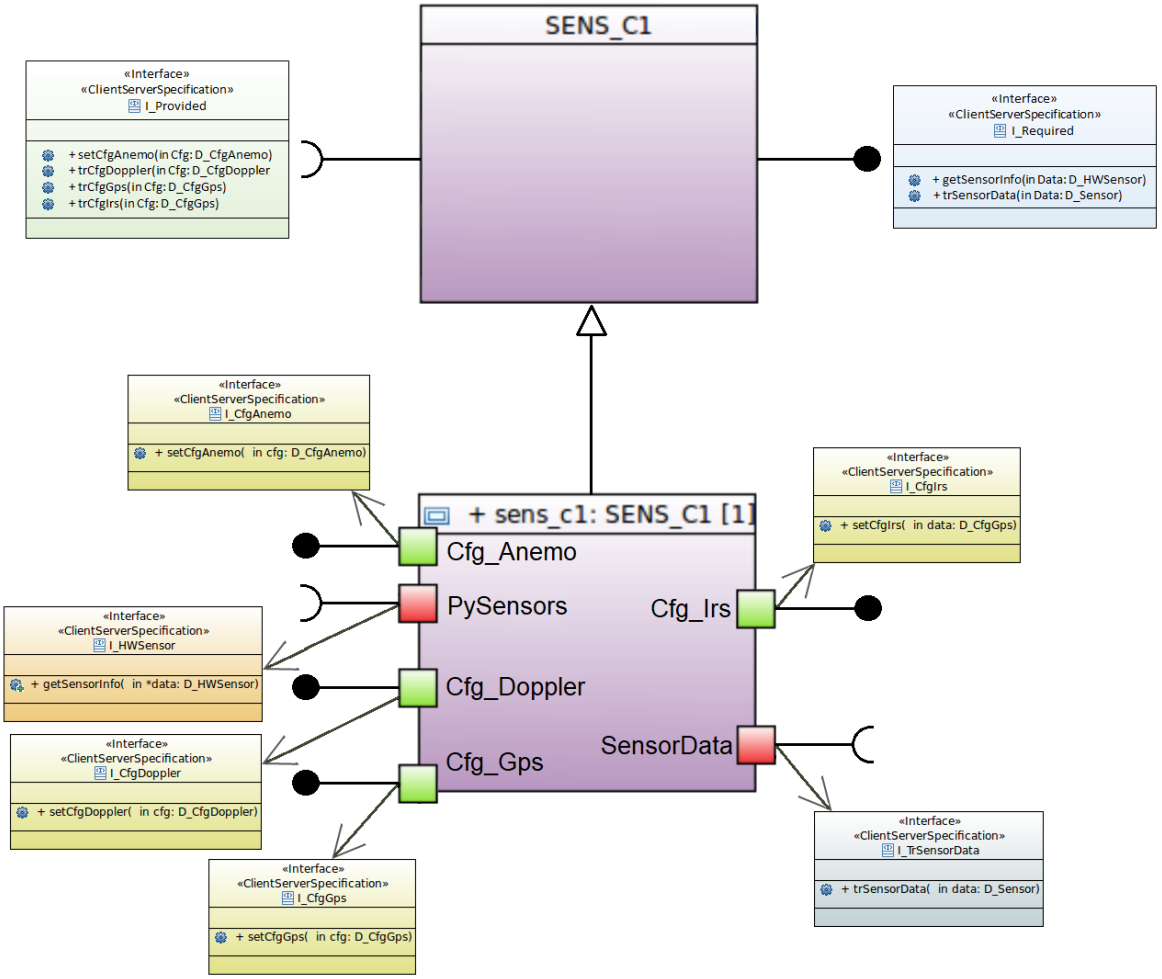


Figure 7 SENS_C1 as a generic and as an inherited, application component.

As it will be seen afterwards, the 'RtUnit' may trigger as many concurrent threads as required. Some of them will be related with the concurrency required by the implementation of the interface functions under the MoC defined. Nevertheless, in an 'RtUnit', there is only one 'main' function. This means that, in principal, apart from the interface concurrency, there is only one active thread per component. If additional threads are required, they could be created as forks from the main thread. In order to ensure that the code is platform-independent, concurrent languages able to be compiled to different platforms should be used, such as C11, Java, ADA, OpenMP, OpenCL, Qt, etc.

Subsystems

Application components can be grouped together in subsystems. A subsystem is just a component. It includes other components inside and, therefore, a subsystem is 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.

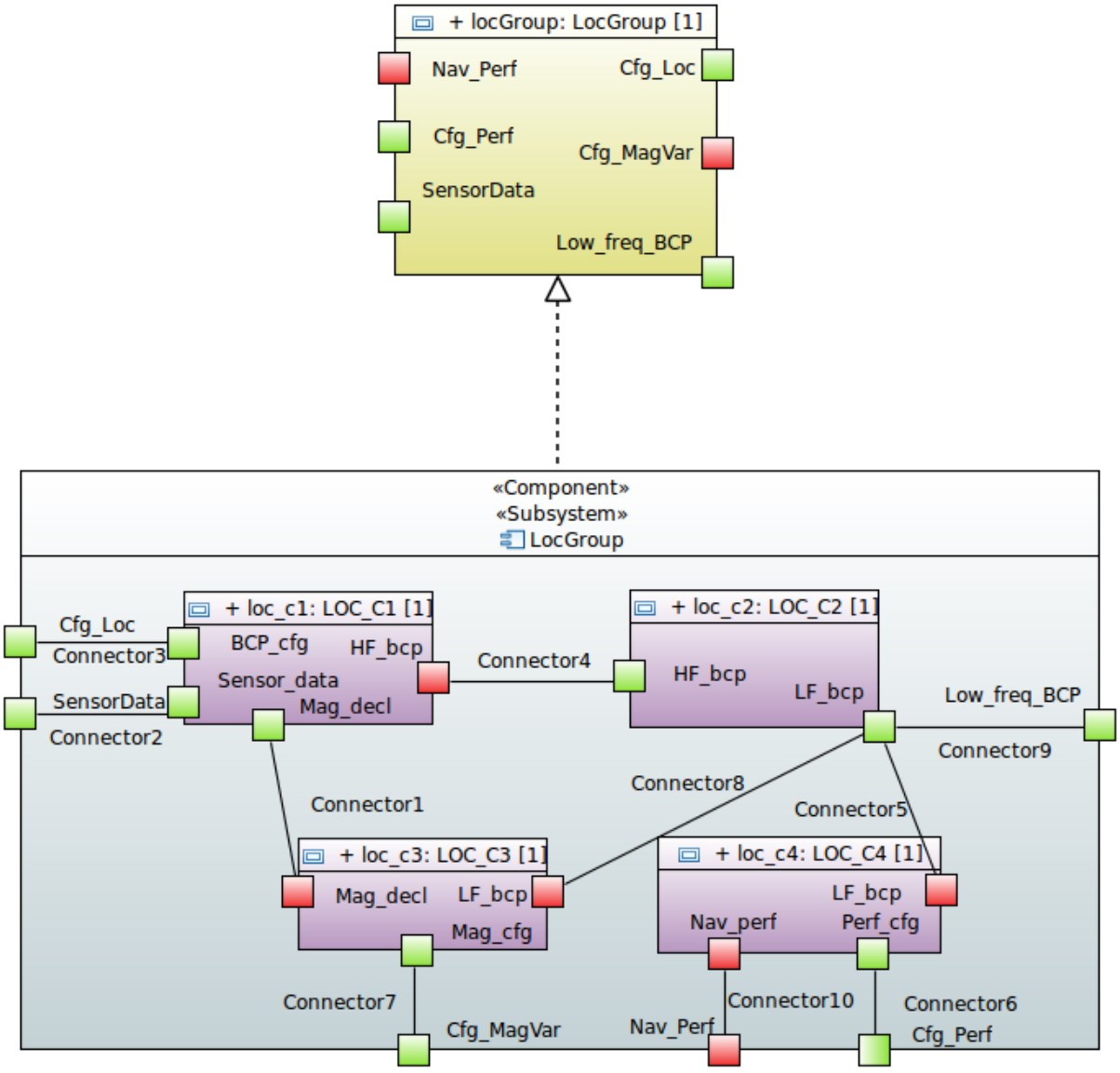


Figure 8 The structural subsystem, locGroup.

If the subsystem does not have any internal functionality and is just a structural architecture of internal components, the subsystem is said to be a 'structural subsystem'. This is the case of the 'locGroup' in the Thales FMS composed directly from the interconnection of components 'loc_c1', 'loc_c2', 'loc_c3' and 'loc_c4', as shown in Figure 8. The global functionality of a structural subsystem comes directly from the composition of the functionalities of their internal components. Otherwise, if the subsystem, apart from the functionality of its internal components has its own internal functionality and resources, it is said to be a 'functional subsystem'. The functionality will be associated to the subsystem in the same way as with any other component, by a linked file where its structured data and behavior is specified in the action language used.

As the interaction between an internal component in a subsystem and the internal functionality of the subsystem, if any, and the rest of components in the subsystem, if any, has to be clearly defined, only application components can be used. Nevertheless, independently of its internal architecture and functionality, a subsystem can be defined as a generic component without concrete ports; just exhibiting the public required and provided services. In this way, again, the flexibility and thus, the reusability of the subsystem is maximized.

Verification

As it was shown in Figure 3, verification is 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 code is ready, concrete test sequences ensuring the correct behavior of the system and its components can be developed.

2.1.2 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.

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.

Network nodes plays in the platform description the same role as the subsystems in the platform independent model. Figure 9 shows the FMS architecture composed of two nodes, the HW in the airplane and the remote data-base in the cloud connected through an airplane-server data-link.

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 in a node are mapped to memory spaces. Only when the complete system is simple enough this intermediate layer can be removed. Figure 11 shows the implementation of the FMS in two executables, 'CriticalSW' and 'FlightPlan'. The airports Data-base is to be implemented in a third executable.

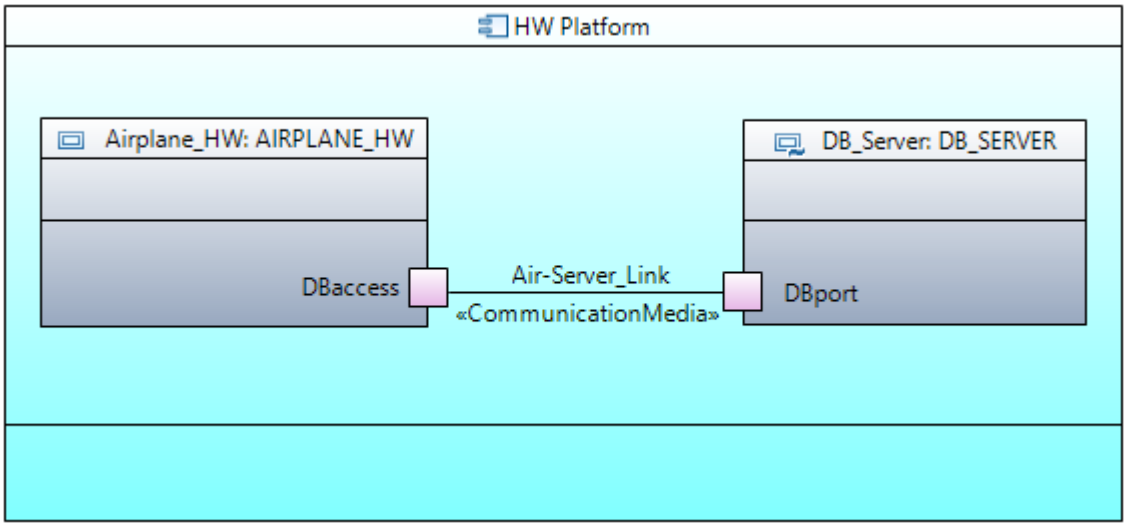


Figure 9 FMS Network architecture.

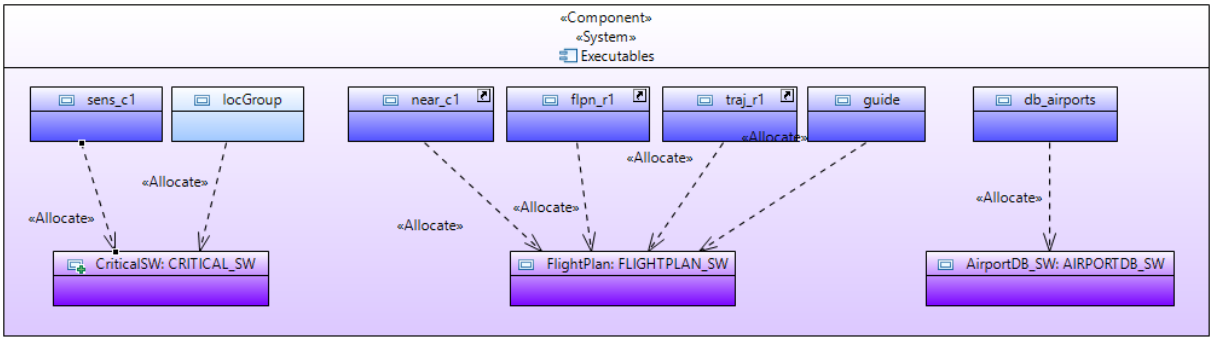


Figure 10 Mapping of Functional components to executables.

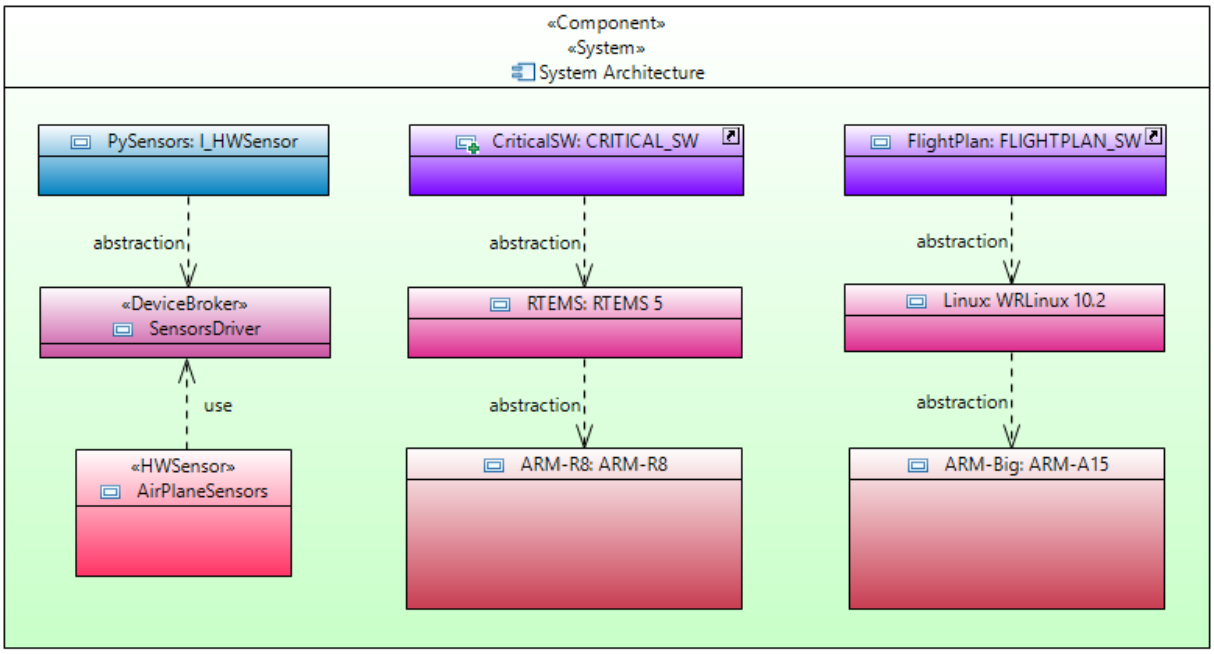


Figure 11 Architectural mapping of FMD components.

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. In Figure 11, the FMS is to be implemented grouping the real-time tasks in an executable on the RTEMS RTOS and the rest of tasks in an executable running on Linux.

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. In the general case, these HdS has two layers. The first one is the device driver. In general, this piece of code is an integral part of the device. As it will be commented later, any HW device should be associated to its IP-XACT model. The IP-XACT model should include the code of the device driver. In order to ensure that the PIM model is really platform-independent, the application code should not call directly the driver functions of any device. Therefore, a second layer is usually required implementing the PIM interface functions making use of the concrete functions of the device driver. 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. In Figure 11, the HdS for the interface 'I_HWSensor' is provided. The 'deviceBroker' provides the implementation of the function 'getSensorInfo(D_HWSensor *info)' using the RTEMS 'driver' for the AirPlaneSensors device.

Hardware resources

MARTE supports the modeling of HW providing a functional classification of hardware entities such as processors, memories, busses, peripherals, etc.[1] They are grouped in the HW modeling package. In S3D the HW logical stereotypes <<HW_PLD>> and <<HW_ASIC>> have a physical semantics and should not be used in the HW resources view. In Figure 12, the computing architecture of the airplane HW is shown. There, a dual ARM Cortex R8 has been used to implement all the real-time functions and a Cortex A35 for the non-critical functions.

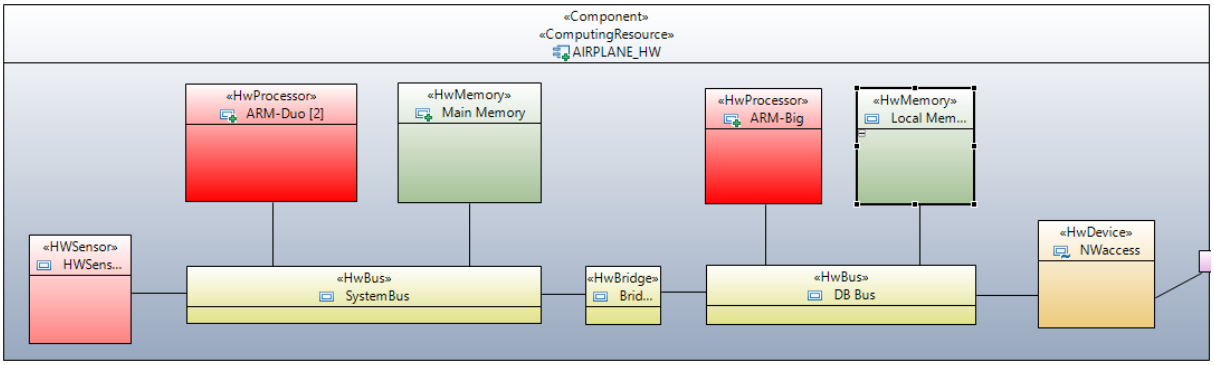


Figure 12 HW architecture for the "Airplane_HW" Node.

Silicon implementation

The MARTE 'HW_Physical' model represents hardware resources as physical components with details on their shape, size, position within platform, 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 Figure 13, the ARM R8 and associated devices are to be implemented in a FPGA, the main memory will make use of a

[1] Computing resources would correspond to the 'devices' in programming languages such as OpenCL.

commercial chip (stereotyped as <<Hw_Component>>) 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.

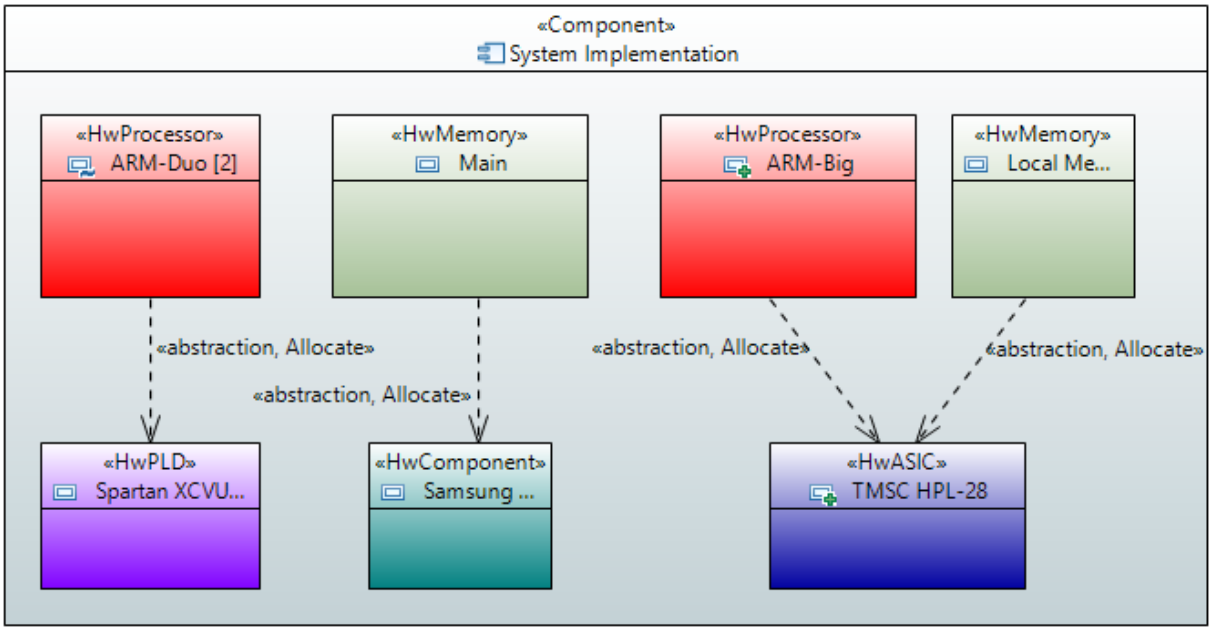


Figure 13 HW implementation.

2.1.3 Platform-Specific Model

The Platform-Specific Model (PSM) captures al the implementation decisions taken during the design process.

Architectural mapping

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. In Figure 11 it is possible to see the design decisions taken for the architectural mapping of the FMS application components.

2.2 D&V Views

As commented above, the complete model is organized in views. Each of these views captures a specific aspect of the system to be designed. The views are modeled as UML packages specified by the corresponding stereotype. The stereotypes, classified by the modeling type, are the following:

PIM views

<<ApplicationView>>

<<VerificationView>>

PDM views

<<MemorySpaceView>>

<<SwPlatformView>>

<<HwResourceView>>

PSM view

<<ArchitecturalView>>

Figure 14 shows how the model is organized in Eclipse EMF Neon. As it can be seen, the model is composed of packages for each of the views commented above. The 'Thales_UC_v2' model of the 'Thales_FMS' project makes use of the legacy components in the 'Thales FMS components' library.

2.3 Components Library

Facilitating reusability is one of the main concerns of the S3D modeling methodology. Reusability requires encapsulating the component in a way that facilitates its reuse. This is the goal of generic components. The second step would be its integration in a reusable component library.

A component is modeled as a package. The package will contain all the relevant information about the component. The first element in the package is the component itself, stereotyped as a 'RtUnit' or a 'PpUnit'. In addition, the minimum information to be provided is represented by the interface functions of the component and the data types used by them. In order to support simulation, performance analysis and synthesis, the files with the code in an appropriate action language are required. Component verification would require a specific package with the test cases in an appropriate language.

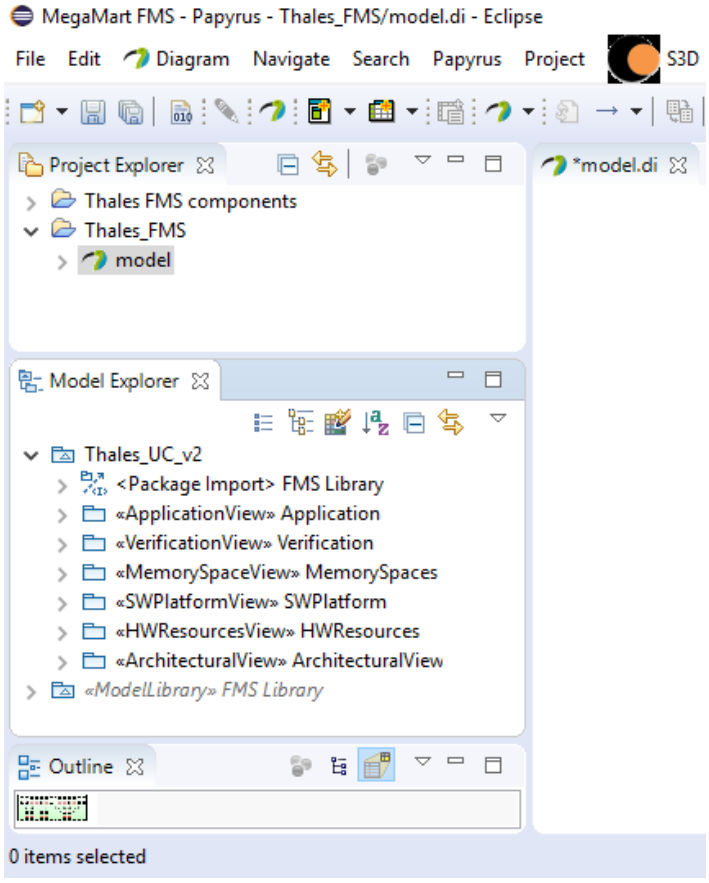


Figure 14 Model views and Component Library.

Figure 15 shows the model of component 'SENS_C1' to be used in the 'Thales_FMS' system. As it can be seen the component 'SENS_C1' is a 'RtUnit' with 'run_sens_c1' as main function. The 'DataTypes'

package includes the data types used by the interface functions. They are included in the 'Interfaces' package.

The files capturing the functionality of the component in C/C++ and Java are described in the 'FileFolders' package. The 'TestData' package includes the test files to verify the component in a GoogleTest framework. A 'Class Diagram' named 'Functionality' is used in order to associate instances of all these files describing the functionality of the component and its verification tests to a 'generalization' of the component.

2.3.1 Active Components

Active application components are modelled as UML components with the MARTE stereotype <<RtUnit>> (Figure 16). An *RtUnit* component has its own execution thread, its associated files with its functionality in an Action Language (i.e. C/C++), and will provide/require services to/from other application components by means of provided and required interfaces. These provided/required interfaces and code files are defined in the *FunctionalView*.

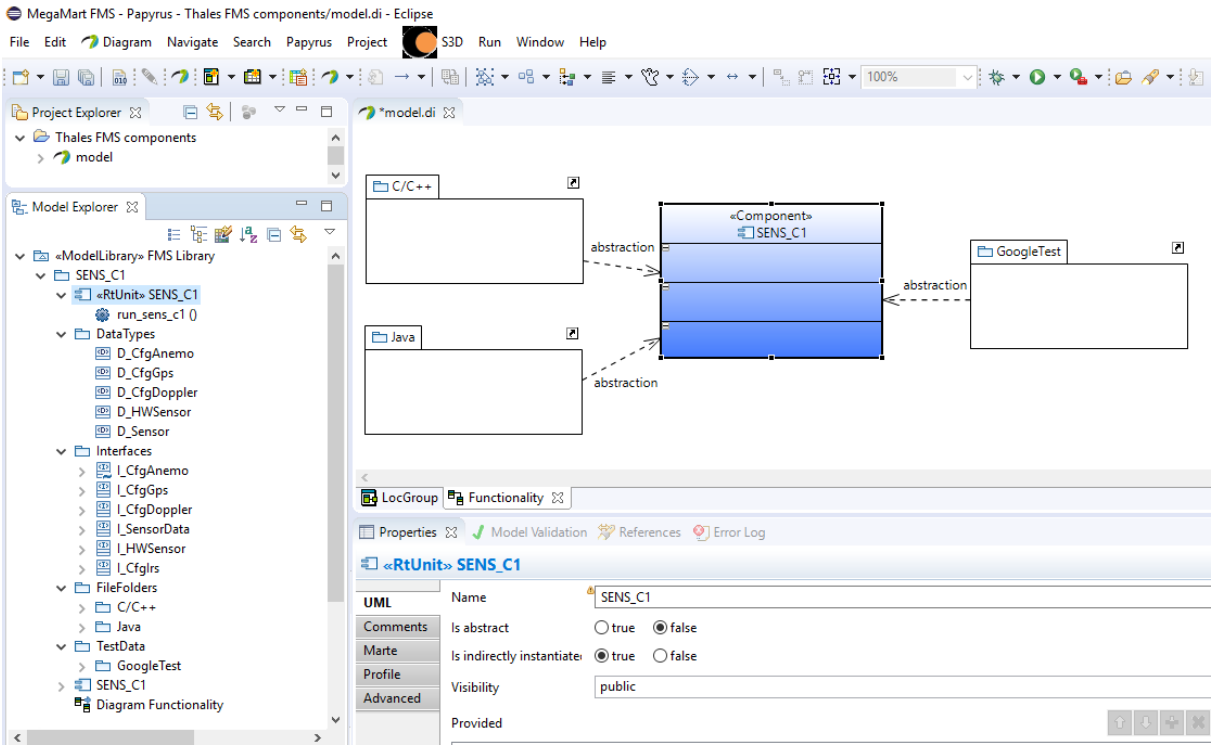


Figure 15 SENS_C1 component.

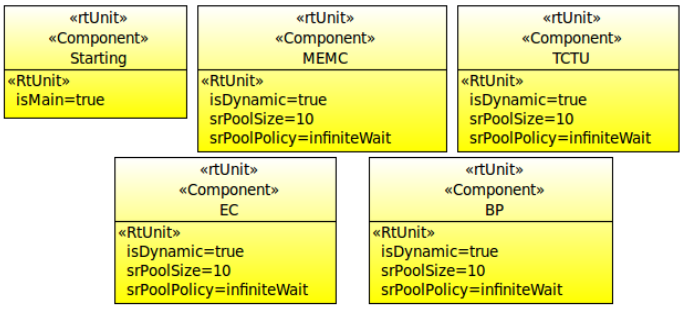


Figure 16 Active application components.

Active Component Attributes

The following attributes of the <<RtUnit>> stereotype are supported:

- The attribute ***isDynamic***. A value *isDynamic=true* specifies that the application component dynamically creates threads in order to attend the requests to the services provided by the *RtUnit*,
- The attribute ***srPoolSize*** specifies that the *RtUnit* has a finite set of threads to attend requests to the provided services,
- The attribute ***srPoolPolicy*** should be *infiniteWait* to denote that, in the event that there is a service request and the RtUnit cannot create a thread to attend the service (because the srPoolSize limit has been reached), the *RtUnit* waits until one of its server threads is released (after completing a service request),
- The ***isMain*** attribute is used to specify that the *RtUnit* has a main thread to be activated when the application executable is launched.

The main function of an active Component

In order to define the main function of the component, the "main" attribute of the <<RtUnit>> stereotype is used. The attribute is assigned to a UML operation captured in the functional view, as shown in Figure 17:

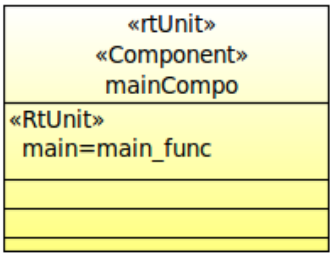


Figure 17 Main function of an application component.

So, the *mainCompo* component will generate a static thread executing *main_func*. S3D recommends avoiding generating dynamic threads from the main function. Nevertheless, if this is needed, it can be done using a concurrent language (i.e. C11, Java, ADA, OpenCL, Qt, etc.).

The main function of a component may have input parameters. In order to annotate these values, a UML constraint is used. The constraint has to be owned by each instance of a component. In the constraint, the name of the functions and the values of their parameters is captured by means of the following syntax: "$initValue=nameFunction(value1,value2,value3)".

2.3.2 Passive Components

Protected passive units (*PpUnit*) are used to model shared information required by active components. a) shows a PpUnit component providing services through three ports. b) shows a PpUnit requiring services. Nevertheless, this happen only as a consequence of the execution of the provided services, not as a consequence of any internal

PpUnits may specify their concurrency policy either globally or for all of their provided services through the *concPolicy* property. The services provided by the *PpUnit* are enclosed in interfaces and

offered through provided *ClientServerPorts*. All the interfaces provided by a *PpUnit* component inherent the value of the attribute *concPolicy*.

As in the case of the *RtUnit* components, Generic PpUnit Components may have associated files, files folder and libraries in order to describe its functionality.

2.3.3 Subsystems

Subsystems are modeled in a similar as leaf components. These components have an internal structure, composed of interconnected application components. The subsystem does not impose the number of ports. They will be decided afterwards, once the subsystem is instantiated in any system application. As it is shown in Figure 18, the 'LocGroup' subsystem follows the same organization than a single, "leaf" component.

The main difference is that the subsystem includes as packages the models of all the components the subsystem uses. Another difference is that the subsystem is stereotyped as a <<Subsystem>>. The figure includes the composite diagram showing the internal architecture of the subsystem. As it is by itself a generic component in the system, it has no ports. This shows how the methodology supports hierarchical partitioning, an essential feature to enable mega-modeling.

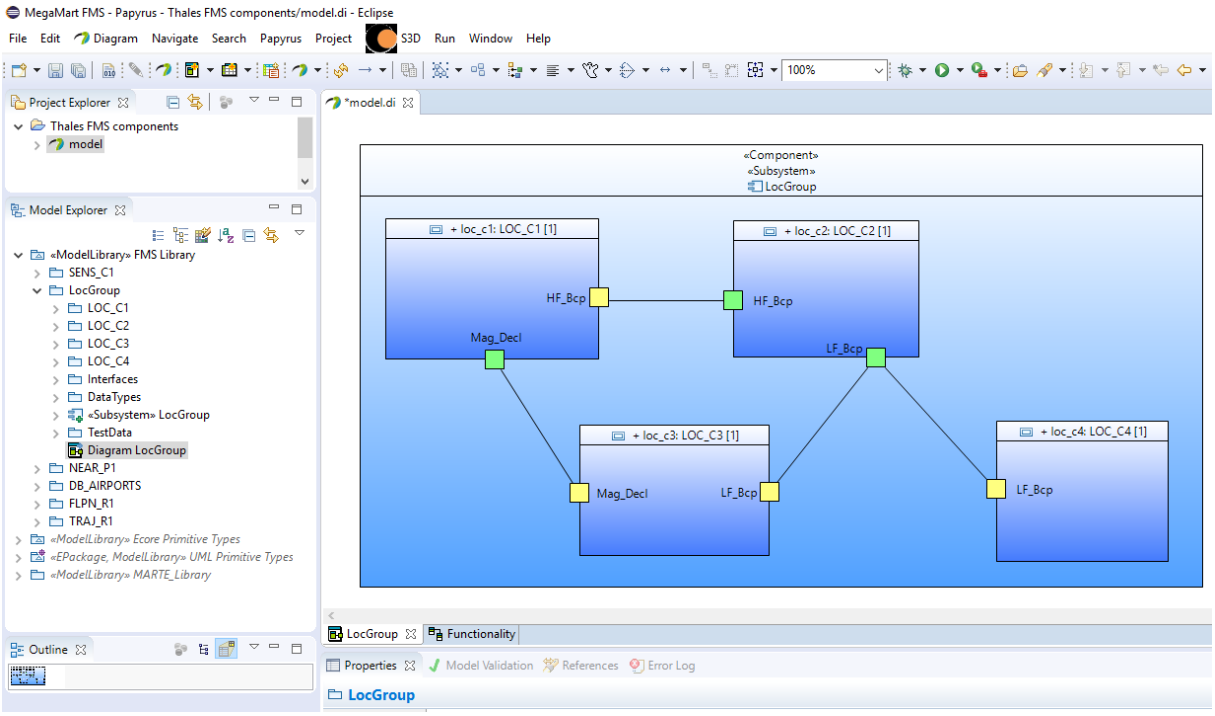


Figure 18 LocGroup subsystem.

2.3.4 Data Types

The UML elements that can be used to define the data types of the system are UML Enumerations (enumerated types), UML Primitive Types (basic data types such as "unsigned char", "int", "long", etc.) and UML Data Types that are used to define new data types.

Enumeration Data types

Enumerations are captured as UML Enumeration data types and the different values of the enumeration are modelled as Enumeration Literals as shown in Figure 19

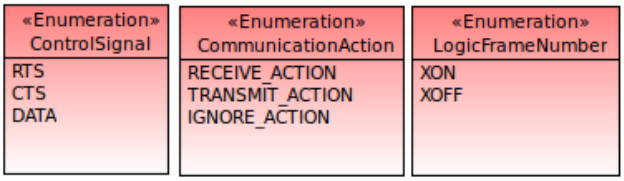


Figure 19 Enumeration data types.

Primitive Data types

UML PrimitiveTypes are used to define basic data types, as shown in Figure 20

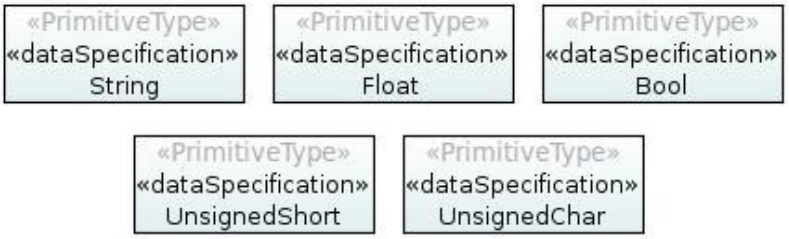


Figure 20 Primitive types.

Derived Data types

The UML DataTypes are used to define new kinds of data. UML Data types are used for modelling non-primitive data types (derived data types). They are structured data and arrays.

Structure Data types

Structured Data are modelled by using the MARTE stereotype <<TupleType>>. The *Datatype* has a set of properties typed by the specific data type or primitive type that represent the fields of the structured data type. When a field of the structure data types is a pointer, an asterisk is annotated in the name as in the "newp_support" data type of Figure 21

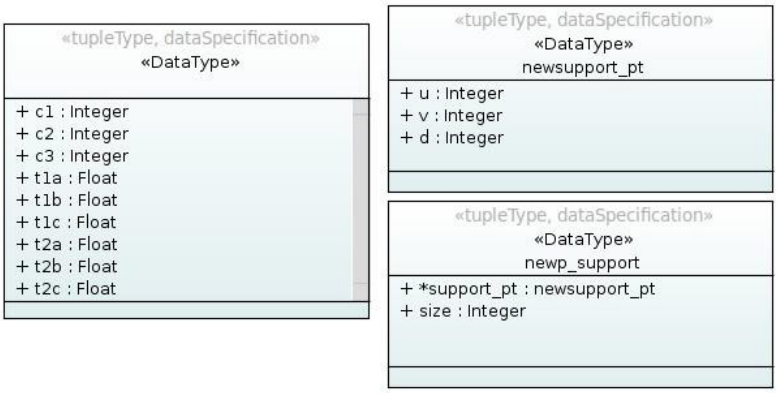


Figure 21 Structure Datatype.

Array Data types

Arrays are modelled by using the MARTE stereotype <<CollectionType>>. The *collectionType* stereotype is applied to a *DataType* model element. A property has to be added to this DataType. The property should be typed by PrimitiveType or another DataType. Then, in the attribute

collectionAttrib of the stereotype *CollectionType* that property should be attached, as property "array128i" in Figure 22.

If the array is unidimensional, its dimension is annotated in the multiplicity tag. If the array is multidimensional, the attribute should be specified by the MARTE stereotype <<Shape>>.The definition of the dimensions is {dim1, dim2, dim3} (Figure 22 and Figure 23). In these cases, the definition of the size (in Bytes) of the array should be annotated as (X,Bytes)x(Y,Bytes)x(Z,Bytes) or by the notation (X*Y*Z, Bytes) (Figure 22).

In some cases, the designer prefers not to specify the dimensions of the array. Figure 24 shows two cases of how to define an array with no specific value of its dimensions. In the case of a unidimensional array, the size is defined in the tag *multiplicity* as [0…*] of the corresponding property of the Datatype. In the case of multi-dimensional arrays (by applying the stereotype *Shape*), the corresponding dimension should be specified by "*". Figure 24 shows examples of these undefined annotations.

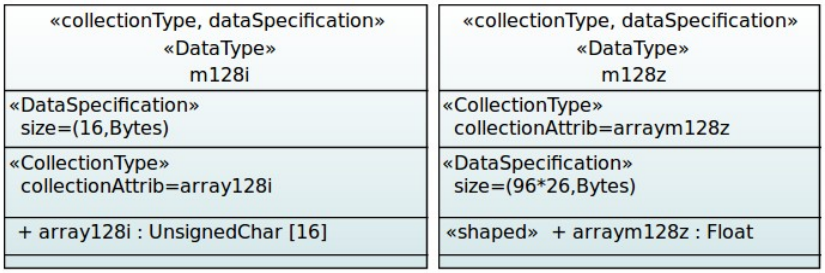


Figure 22 Array modelling.

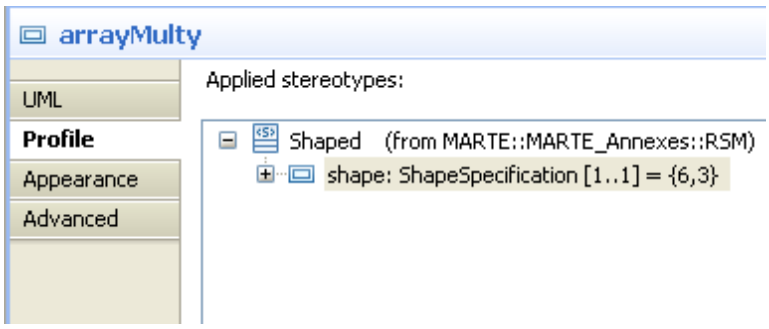


Figure 23 Array dimension specification by the Shape stereotype.

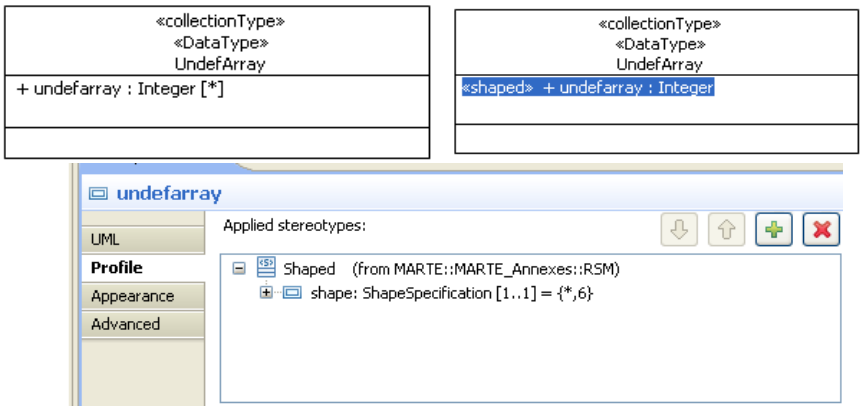


Figure 24 Arrays with undefined dimensions.

Completely specified data types

The methodology includes a stereotype for completely specifying the data types. The attributes associated with this stereotype are shown in Table 1:.

<<DataSpecification>>
size: NFP_Data [1]
pointer: Boolean [1]
dataSpecifier: DataSpecifier [1]
dataQualifier: DataQualifier [1]
complexDataType: String [0..1]

Table 1: <<DataSpecification>> stereotype attributes.

The attributes are:

size: defines the size of the data in its memory representation. The attribute size is NFP_Data, a MARTE data type that specifies the size of a data. The notation of this MARTE type consists of two values, the value and the unit. It can be annotated in two different ways:

- size: NFP_DataSize[1] = (value=8, unit=Byte), where the value is a real number and the unit might be bit, Byte, KB, MB or GB.
- size: NFP_DataSize[1] = (16,Byte).

pointer: specifies whether the data is a pointer

dataSpecifier: denotes the data type. In order to ensure language-independency, this attribute is defined by a *string*. So, the list of values in the case of the C/C++ language is the following:

<<String>> DataSpecifier		
None	Int	long
char	signed int	long long int
signed char	unsigned	signed long long
unsigned char	unsigned int	signed long long int
short	long	unsigned long long
short int	long int	unsigned long long int
signed short	signed long	float
signed short int	signed long int	double
unsigned short	unsigned long	long double
unsigned short int	unsigned long int	void

Table 2: Data Specifier Values.

dataQualifier: denotes the data qualifier. In the case of C/C++, the list of values of the *DataQualifier* attribute is the following:

<<String>> DataQualifier
none
const
volatile
register

Table 3: Data qualifier values.

complexDataType: can only be used when the possible values of the *dataSpecifier* and *dataQualifier* cannot specify the data type. For instance, “complexDataType = const volatile unsigned long int”.

2.3.5 Generalization of Data Types

The modeling methodology enables the definition of data types. This is modeled using the UML inheritance. If the parent element of the UML inheritance is a Primitive Type (in Figure 25, the data “ULONG” and “USHORT”), the specific data is specified by the values of the corresponding primitive type captured in the attributes of the stereotype *DataSpecification* (the attributes dataSpecifier or the *complexDataType*). If the parent element of the UML inheritance is a Data Type (in Figure 25, the data “Byte”) the specific data is specified by the DataType (in Figure 25, the “QoS” is specified as “BYTE”).

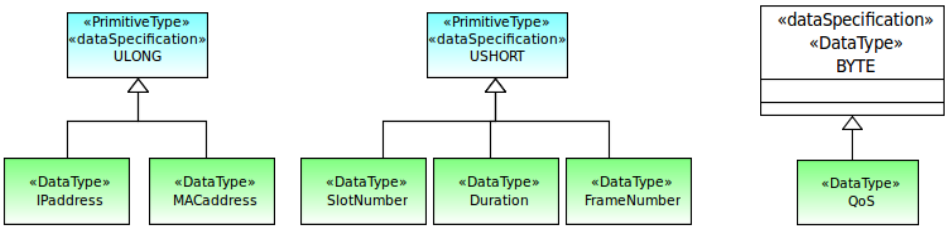


Figure 25 Data Generalizations.

In some design optimization tasks, it is required to divide a data type containing many data in a new, related data types, with a subset of the data. In these cases, data type inheritance can be a solution. Some modelling constraints are applied to these data type inheritances:

- Both data are of an UML Data Type,
- The stereotype *DataSpecification* should apply to both data types,
- The attribute *complexDataType* of the *DataSpecification* stereotype of the derived data type (in Figure 26, the Data Type *array_memc_tctu_exploration*) should be specified by the name of the parent data type (in Figure 26, the Data Type *array_memc_tctu*),
- In the attribute *size* of the *DataSpecification* stereotype, the new and different value of the size (in Bytes) of data should be specified:

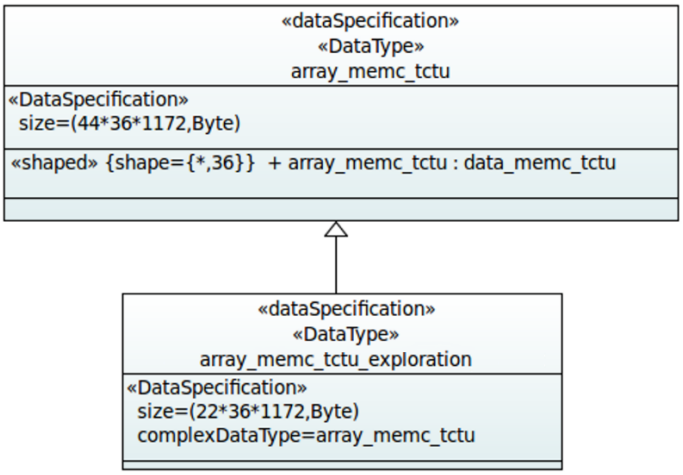


Figure 26 Data Type generalization.

In this way, a service using *array_memc_tctu* can be provided by executing two times the same service using *array_memc_tctu_exploration*.

2.3.6 Files

The files that store the implementation source-code of the applications are modeled as UML artifacts. These artifacts are specified by the UML standard stereotype <<File>>. The Artifacts are specified by a name (annotated in the attribute "name") and in the attribute "File name", where the name and the extension of the file should be included, as shown in Figure 27:

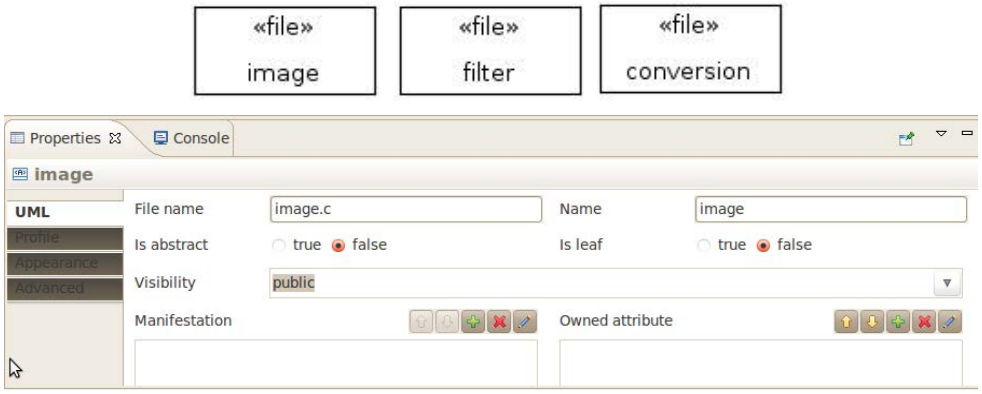


Figure 27 Files.

File specification

Each File can be specified in more detailed with additional information. This additional information is captured in the stereotype <<ApplicationFile>>. The *ApplicationFile* stereotype has the following attributes:

parallelized: Boolean. The file contains code in a concurrent language (i.e. C11, OpenMP, OpenCL, Qt, etc.) leading to several threads when executed,

highLevel: Boolean. The file corresponds to a high-level language which cannot not be compiled directly (i.e. Heptagon from which C can be obtained),

implementation: String. The file is optimized to be executed in a specific HW resource: DSP, NEON, GPU, etc. The name annotated should be the same as the HwISA of the HW processor specified in the HwResourceView used for the allocation.

notModifiable: Boolean. The file is protected and cannot be modified.

environment: Boolean. The file corresponds to a test bench of the system.

<<ApplicationFile >>
parallelized: Boolean [1]
highLevel: Boolean [1]
implementation: String [0..1]
notModifiable: Boolean [1]
environment: Boolean [1]

Figure 28 ApplicationFile stereotype attributes.

Association of Files, File Folders and Libraries to Generic Components

The association of files, file folders and libraries to Generic Components is specified using a UML Class diagram. The association is made using the UML connector <<use>> between instantiations of the files in the file folder and a generalization of the component, as shown in Figure 26.

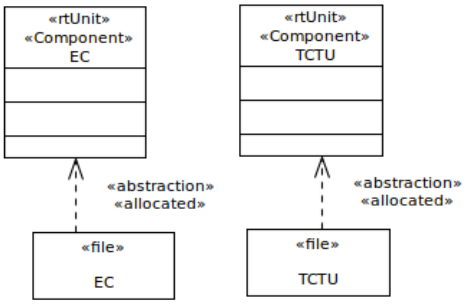


Figure 29 Association of files to Generic Components.

2.3.7 Interfaces

Interfaces integrate the services provided/required by a component and define its characteristics. Each interface function is stereotyped as a Real-Time service ('RtService'). Its properties are defined by the attributes described in Section 3.3.1. All the functions included in the same interface share the same properties. The same function can be included in different interfaces with different properties.

Interfaces are modelled by means of UML interfaces. UML interfaces are stereotyped by MARTE <<ClientServerSpecification>>. A *ClientServerSpecification* provides a way to define a specialized interface that is to be defined in terms of its provided (or required) operations.

Generic Interfaces

In principal, a Generic Component has as many interfaces as provided/required services. Only when a set of services are going to be handle together always, they are grouped in the same Generic Interface. In order to maximize reusability, these Generic Interfaces should not be annotated with properties limiting its architectural applicability. When a generic component is used in a concrete

architecture, its ports should be associated with the corresponding application interfaces including all the functions required or provided by the port. Application Interfaces will integrate services of several Generic Interfaces as required by the connectivity of the application component in its use in the application architecture.

Application Interfaces

When a component is instantiated in a system (or subsystem) architecture, the ports through which the component communicates with the other components in the system (or subsystem) and with the environment are decided. All the services required (or required) by the port should be grouped in a single interface. This interface is defined by inheriting all the interfaces whose services have to be grouped, as shown in Figure 30:

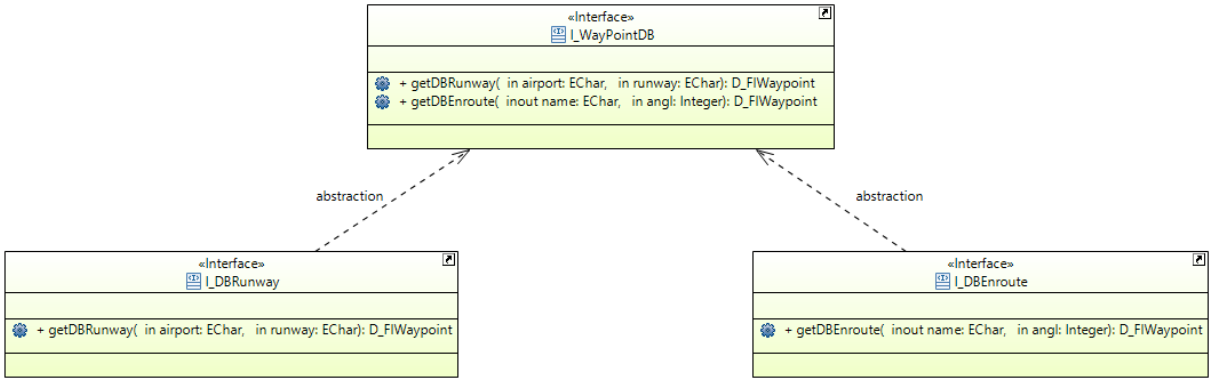


Figure 30 Application Interface based on two Generic Interfaces.

Interface Services

Interface functions are stereotyped as Real-Time services (*RtService*). Their properties are defined by several attributes. Due to its importance in order to define the MoC of the component and its environment, they are described afterwards, in Section 3.3.1.

Service Arguments

In the general case, Interface Functions have arguments. These arguments are modelled as UML parameters. These parameters ca be typed by the Data types defined in the Data Model. The UML parameters can be **in**, **inout** and **return**. The order of the arguments in a function prototype has to be specified. For that purpose, the name of the UML arguments that model the function arguments should be defined as **order:nameArgument** where the value **order** defines the order of the argument in the function prototype.

Pointer

The function arguments can be modelled as pointers by applying the stereotype <<Pointer>>.

Reference

The function arguments can be modelled as references by applying the stereotype <<Reference>>.

Arguments Qualifier

The function arguments can be specified by a qualifier by applying the stereotype <<ParameterQualifier>>. Values associated with the *ParameterQualifier* stereotype are "const", "volatile" and "register".

Interface compatibility

The methodology supports interface compatibility. Interface compatibility enables the connection of components with different interfaces whenever the service required by one component can be realized by the service provided by another component. Thus expanding reusability. Moreover, interface compatibility allows exploring different design alternatives based on different concurrency levels.

In order to be compatible, two interfaces have to share at least one compatible function. These compatible functions are the functions to be provided by one component and required by the other. Compatibility between interfaces and interface functions can be expressed using UML inheritance in the same way as with data generalization (see §2.3.5). The parent function is provided while the inherited functions are required by one or several components.

Two functions with different names and parameters are compatible whenever:

1. the set of parameters of the derived function is a subset of the parameters of the parent function,
2. the types of the parameters in the subset are the same or a generalization of the parameters in the parent function and with the same direction,
3. all the parameters of the parent function are associated once and only once to the parameters of the parameter sets of all the inherited functions. Parameters in the parent and the inherited functions should be associated unambiguously either by name or type. Of course, the direction must be the same,
4. an *inout* parameter in the parent function may be associated to two parameters in one or two inherited functions provided that such association is unambiguous,
5. independently from/to which inherited function a parameter is got and eventually, changed, the functionality performed is that of the parent function.

In Figure 31, the service *getDBRunway* can be provided whenever the services *getAirport* and *getRunway* are required and the two-input data needed (*airport* and *runway*), got from them. The result will be provided when the service *getRunway()* is required. Parameter *airport* is associated by name while parameter *runway* is associated by type.

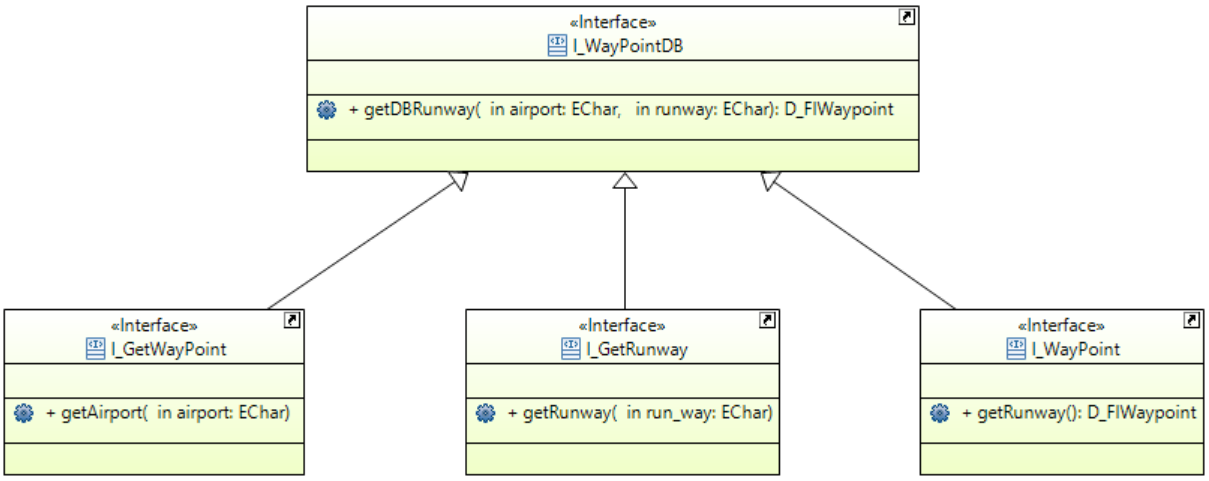


Figure 31 Interfaces inheritance.

Interface compatibility is very useful in breaking the sequentiality imposed by services with input and output data. In general, it is difficult to avoid stopping the execution of the thread requiring the service until it is executed in the provided component and the results (*inout* and *return* parameters), produced. This limitation is particularly important in signal processing systems. With interface compatibility, the times in providing and delivering data are different which means that the service can be called as frequent as needed, independently of its execution time (whenever the throughput is kept).

2.3.8 Libraries

In order to compile an application, it is necessary to include all the libraries used when developing the code. Therefore, in order to enable the generation of the makefiles, these libraries should be modeled. Libraries are modeled as UML Artifacts specified by the UML standard stereotype <<library>> as shown in Figure 32:

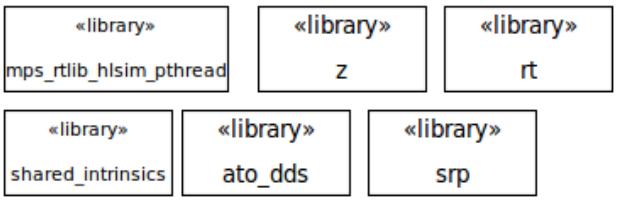


Figure 32 Libraries

2.3.9 Auxiliary Files

As was described previously, each application component has the files that implement each specific application functionality associated. However, these files can require functions that are implemented in other files and which act as auxiliary files that provide services for the application functionalities. These auxiliary files are modeled as UML packages in order to represent the folder where these files are allocated. These files are specified by the stereotype <<FilesFolder>>.

The *FilesFolder* stereotype has the following attributes:

- **parallelized**: the file folder contains files containing concurrent code,
- **highLevel**: the file folder contains files that specify high-level functionality,
- **implementation**: the file folder contains files which are optimized to be executed in a specific HW resource (i.e. DSP, NEON, GPU, etc.),
- **notModifiable** : the file folder contains files which cannot be modified for any reason,
- **environment**: the file folder contains a test bench.

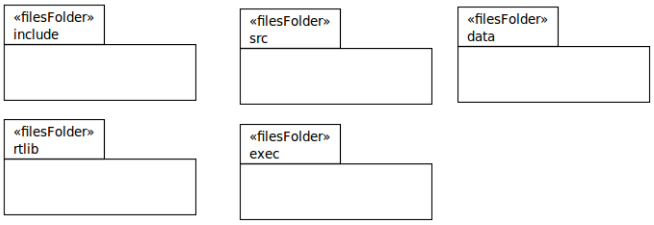


Figure 33 Auxiliary FilesFolder packages.

2.4 Application View

This view allows capturing the generic components selected/developed to implement a system, the application architecture where these components have been instantiated as application components and the way they have been connected each other. A hierarchical approach is used to capture the application model. So, the complete application is captured as a component, the system, which in turn can be hierarchically partitioned in simpler subsystems and components. Three types of components are supported, active, passive and composite components. An application component communicates with other components through client-server ports. These ports have associated required/provided interfaces. Provided interfaces declare the functionalities implemented by the component and offered to other components. Required interfaces declare the functionalities invoked by the component but implemented by others. The application view serves to declare and define these components and to interconnect them, eventually generating the "top" application component, called the system in the application view context. The system component (and by extension, a composite component) is described through the instantiation and interconnection of declared generic components. All these instances and interconnections configure the application architecture. Application components are interconnected each other through port-to-port connectors.

The functionality of the application is derived from the source code in the files associated to the generic components from which the application components have been instantiated. Additional functionality will be derived from the properties assigned to the interfaces and ports defining the way the component interacts with other components and/or the environment. In any case, the application model shall be platform-independent.

2.4.1 Components

In general, components in the Application View are generalizations of generic components and sub-systems in the reusable libraries used in the development of the system. These generalizations will be used in composing the system architecture. Components and sub-systems are connected each other through ports.

2.4.2 Ports

Communication among application components is established through UML ports. The ports link to the interfaces containing the services that the application components require and/or provide. These ports and interfaces may be assigned with properties. These properties would define the model of computation and communication among components.

The ports of the components should be specified by the MARTE stereotype <<ClientServerPort>>. In the attribute kind of the *ClientServerPort* stereotype, the port is specified considering whether the port provides or requires an interface. The interface required or provided by the port is defined in the attributes *provInterface* and *reqInterface*.

In order to specify the interaction properties of the interface, the S3D stereotype <<ClientServerQueuePort>> should be used. This way, system modeling under different MoCs is supported as addressed in section 0.

2.4.3 Connectors

Ports are connected among them using UML connectors.

In Digital Signal Processing applications, the flow of data among components is very relevant in order to understand the behavior of the system. In order to highlight the direction of data movements,

when all the parameters in all the functions in an interface have the same direction, instead of a connector, the two ports may be linked with an *InformationFlow* edge making data direction explicit.

2.4.4 Application Architecture

The top application component is captures as a UML component decorated with the <<System>> stereotype. Within the application view context, this is called the *System* component. Only one *System* component should be defined within the *ApplicationView* package.

The System component is built up with instances of the application components interconnected through connectors. The application architecture is captured in a UML Composite Structure diagram associated with the System component.

2.4.5 System ports: I/O communication

The System component communicates with the external environment. This environment communication is established through ports. These UML ports should be specified by the MARTE stereotype <<ClientServerPort>> (Figure 34), specifying the correct values of the attribute kind, *provInterface* and *reqInterface*. These System ports are connected to application instances. This connection is port-to-port.

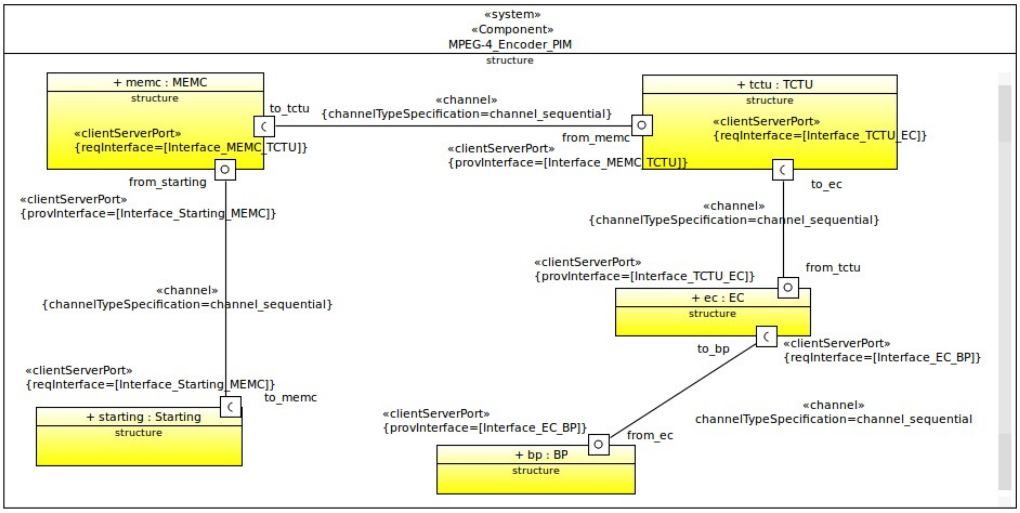


Figure 34 Application Structure.

2.4.6 Periodic Application Instances

The main function of an application component may be characterized by a period, triggering its execution periodically. The period of an application component is modelled by a UML comment specified by the MARTE stereotype <<RtSpecification>>. In the attribute *occKind the* period is annotated as:

- periodic (period= (value, unitTime))

Then, the *RtSpecification* comment is associated to the *RtUnit* instance component by using a UML link (Figure 35):

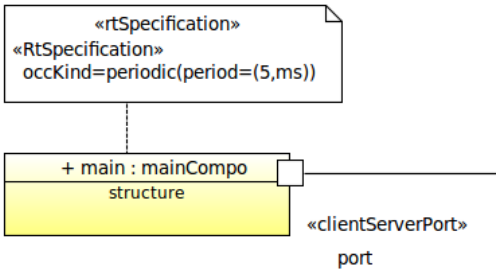


Figure 35 Periodic application instance.

2.4.7 System Files

The System component may have associated files. These files are identified by the UML standard stereotype <<File>> and by the stereotype <<SystemFile>>. These files are associated with the System component through a UML abstraction specified by UML *Use* relations, as shown in Figure 36:

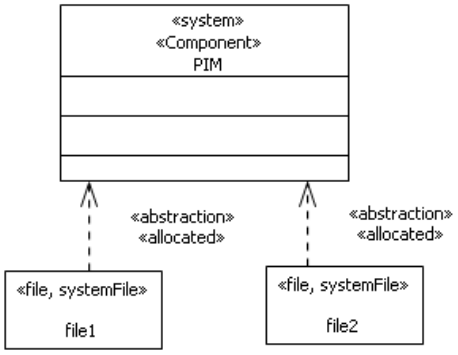


Figure 36 System component with associated files.

Libraries

The compilation of the application may require a set of specific libraries in order to enable the generation of the required makefiles. The *Libraries* defined are associated with the *System* component by means of UML *Use* relations, as Figure 37 shows:

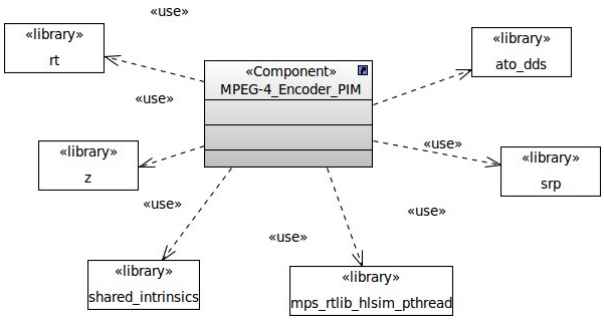


Figure 37 System component with associated libraries.

Files Folders

The *FilesFolders* packages are associated with the System component by UML Use relations. The designer is free to include the corresponding UML artifact files in these packages in order to model the real auxiliary files explicitly.

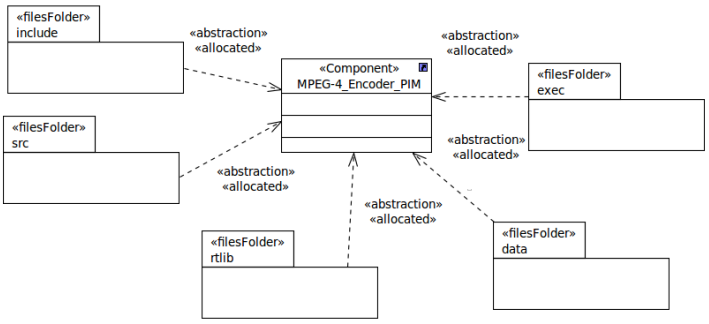


Figure 38 System component with FilesFolder packages.

Modeling Variables

In S3D, modelling variables are used to define characteristics required to fully model the application components of the system in relation to certain deign tasks such as compilation and code generation. The modelling variables are:

- **language**: specifies the language in which the specific application functionality is implemented. Not mandatory (by default, it is "C").
- **path**: specifies the path where the functional files are allocated in the host. Mandatory for the System component.
- **path_system**: specifies a path of a File or FilesFolder of an application component that has as first part of the absolute path, the path associate to the System component.

Modeling Variable Specification

Modeling variables are annotated as *$nameVariable="valueVariable";* as shown in Figure 39:

```
MAC_LMAC_variables
{$language="c++";
$path="yaw/components/mac/";
$creation="ComponentCore";}
```

Figure 39 Specification of Modeling Variables.

The model variables are annotated with UML Constraints owned by the component (RtUnit, System, etc.) denoted in the *ownedRule* of the component and in the "Context" attribute of the constraint (Figure 40).

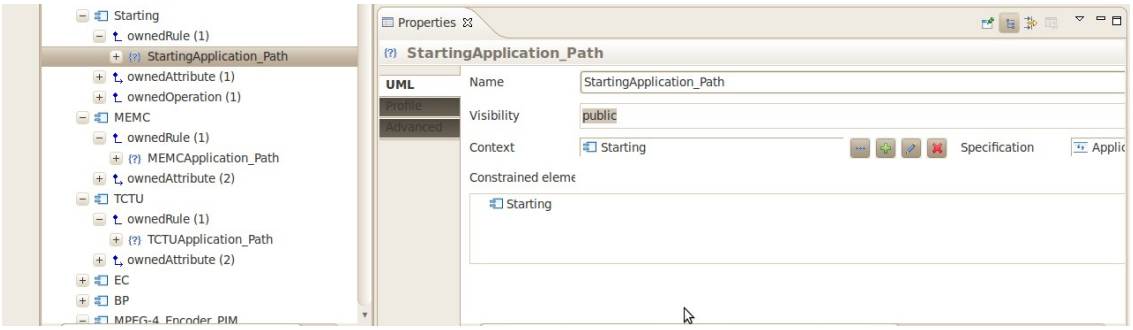


Figure 40 UML constraint for application component variables.

The “Specification” attribute of constraint contains the declaration of the variables. The variable annotation is captured in a *LiteralString* (Figure 41). Then, the constraint is associated with an element model that is included in the *ConstrainedElement* attribute of the UML constraint (Figure 40). The *ConstrainedElement* attribute denotes the model element which the variables annotated in the constraint are applied. This association is captured by using and UML link between the constraint and the model element. It is necessary to distinguish which element is the owner of the constraint and the element to be specified by the variables of the constraint. So, in Figure 41, there are four constraints (“MAC_LMAC_states_facets”, “MAC_LMAC_varibles”, “MAC_InterfacesFolder_LMAC_common” and “MAC_Folder_LMAC”). All these UML constraints are owned by the application component “lmac” (Figure 43). However, not all of these constraints are applied to the same model element, denoted by the attribute “ConstrainedElement” of the constraints (Figure 44).

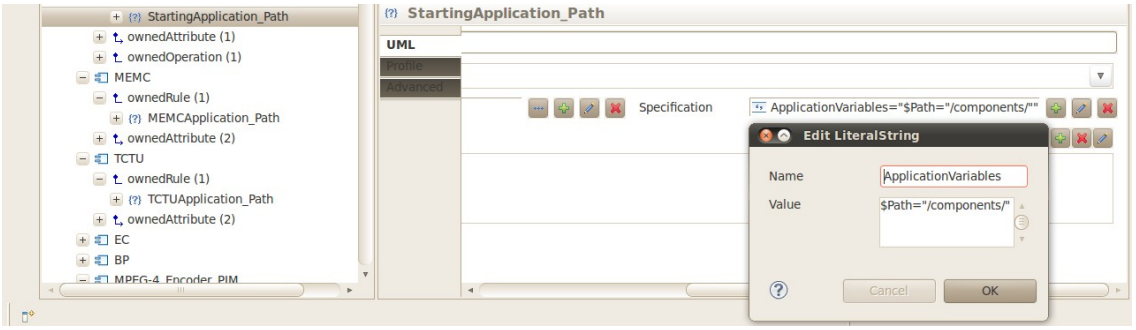


Figure 41 Annotation in a UML constraint for variable specification.

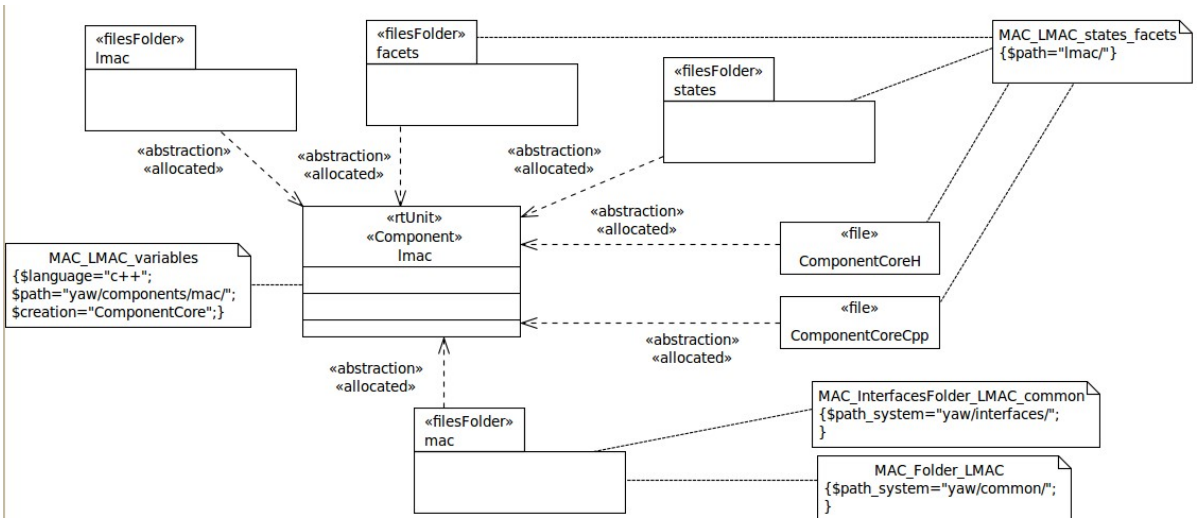


Figure 42 Multiple constraints in the same application component.

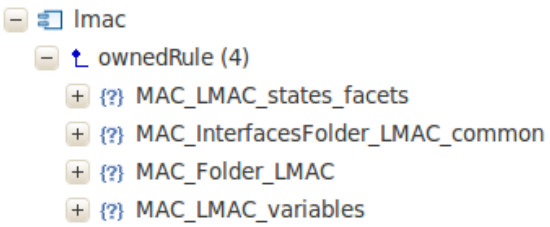


Figure 43 Constrains to the “lmac” application component.

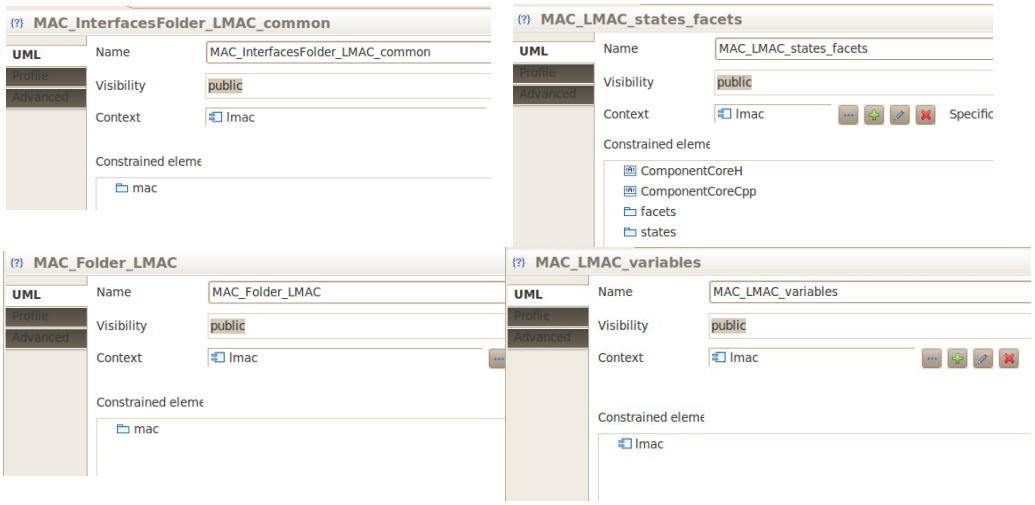


Figure 44 Constraints with different constrained elements.

2.4.8 Concatenation of paths

The creation of the *makefiles* from the information captured in the model requires the paths of the different model elements to be exact. The criteria for composing these paths is a concatenation of different paths. The base path is the *$path* annotated in the *System* component. This path is used to create the complete paths of the different files, file folders, etc. of the application (Figure 45):

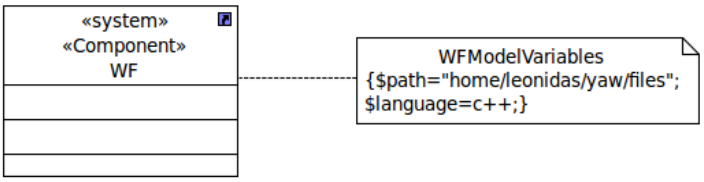


Figure 45 Specification of the System's base path.

Then, each application component has its own relative path. In Figure 46, the application component "lmac" has the associated constraint "MAC_LMAC_variables". This constraint specifies the $language, $creation and $path. In relation to the $path, the base path for the files and files-folder associated with this component is "home/leonidas/yaw/files/components/mac/" that is, the concatenation of the System's base path and the application component path.

To complete the path of the files "ComponentCoreH" and "ComponentCoreCpp" in Figure 46, to the previous path ("home/leonidas/yaw/files/components/mac/"), the path associated with the Files is concatenated as well: "home/leonidas/yaw/files/components/mac/lmac/". Finally, the name of the attribute "File name" of the File model element (see section 3.1) is concatenated. Thus, the path of the File is "home/leonidas/yaw/files/components/mac/lmac/ComponentCore.h".

In the case of the FilesFolder "lmac", it does not have any constraint associated. In this case, the path is the System path (Figure 45) plus the application component path (Figure 46) and the name of the FileFolder (or File): "home/leonidas/yaw/files/components/mac/lmac/".

A different case is the specification of the path for the path "mac". This path has an associated constraint where a $path_system variable is annotated. In this, the creation of the path does not consider the base path of the application component (in Figure 46, "yaw/components/files/"). In this case, the

System path (Figure 46) is concatenated with the value of the $path_system variable and the name of the FilesFolder:

"home/leonidas/yaw/files/yaw/interfaces/mac/" and

"home/leonidas/yaw/files/yaw/common/mac/".

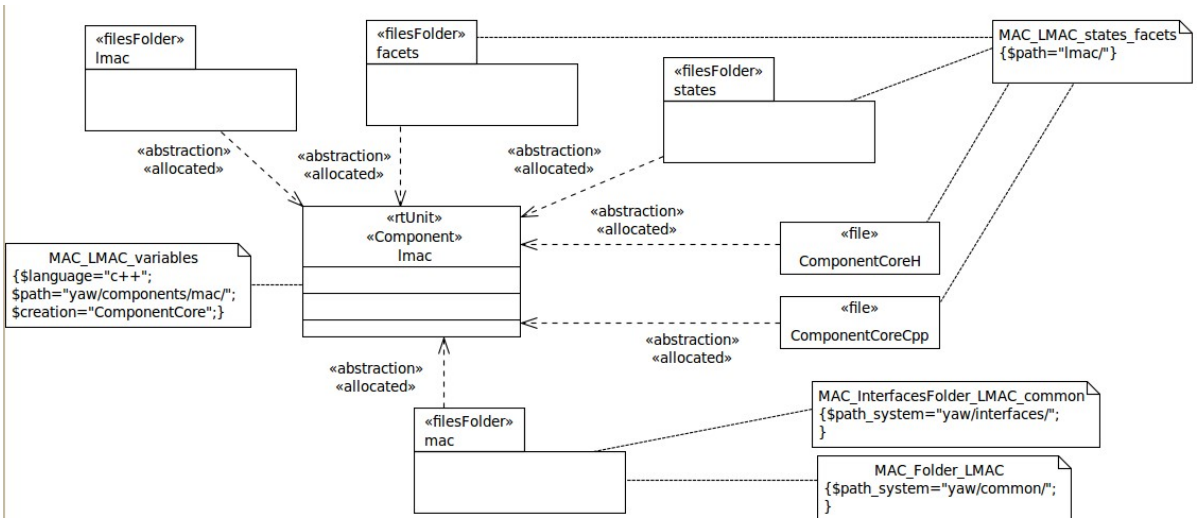


Figure 46 Application components with different types of model variables.

When two or more constraints are associated with a File or FileFolder, this means that there are two or more Files or FilesFolders with the same name but in different locations (in Figure 46, "mac" FilesFolder).

2.5 PDM Views

The PDM views describe the platform on which the system application is going to be mapped. It starts with the memory partitions, that is, the executables in which the components are grouped, the OSs running these executables, the HW resources executing the code and even, the physical devices on which these HW resources have been and/or are to be implemented.

2.5.1 Memory Space View

The memory space view contains the components that identify the memory spaces, which represent the executables of the system. Thus, an executable is a memory space in this methodology. These memory partitions are used for grouping application components. The UML elements used in this view are:

- UML Component for modeling the memory partition types and other Components in order to define executables,
- UML Generalization for relating the System component of the *ApplicationView* with the System component of the *MemorySpaceView*,
- UML Abstraction for associating application components to memory partitions.

Class diagrams are used for defining the memory partition types and for capturing the UML generalization of the *System* components.

Composite structure diagrams are used for defining the memory partition instances.

Process modelling

Memory partitions are modeled by the MARTE stereotype <<MemoryPartition>> applied on a UML component (Figure 47):

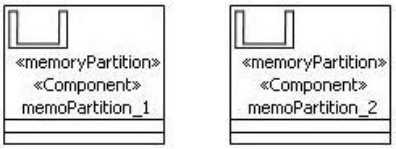


Figure 47 Memory partitions.

The executables are defined in a *System* component included in the view as instances of the *MemoryPartition* components previously defined (Figure 48):

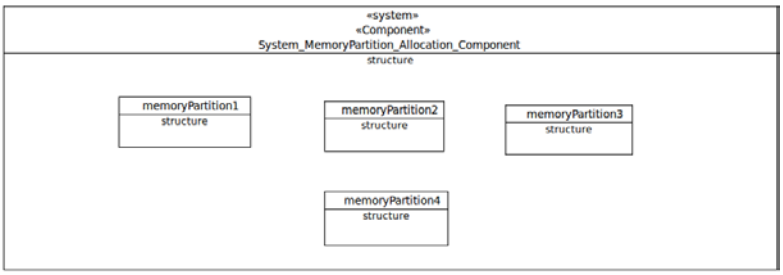


Figure 48 Executables definition.

This system component is used in order to allocate the application instances defined in the *ApplicationView* to the corresponding memory partitions. This System component should be specialized by the System component defined in the *ApplicationView*. This specialization is modelled by means of a UML generalization defined in a UML class diagram. Only one System component should be defined within the Memory Space View package (Figure 49):

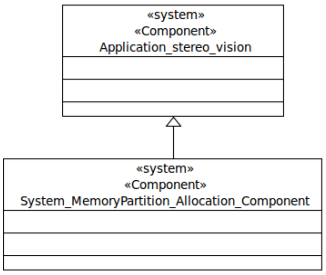


Figure 49 Specialization of the System component of Memory Allocation View.

By means of a UML composite structure diagram associated with the System component, the application instances defined in the System component of the *ApplicationView* are mapped onto the memory spaces. The application component instances are mapped onto memory partition instances by means of UML abstractions specified by the MARTE stereotype <<allocate>>. So, in Figure 50, the yellow boxes are application components that are mapped onto memory partitions.

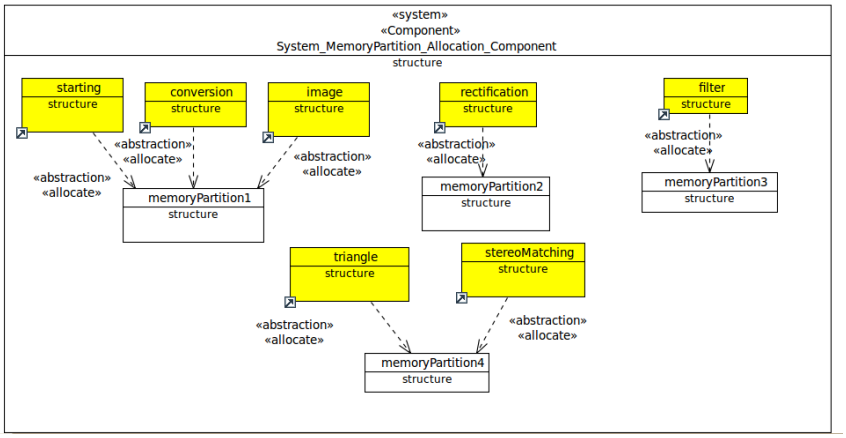


Figure 50 Memory partition allocation.

Composite components Allocation

When an instance of a composite component is allocated in a memory partition, all the internal instances of such composite component are assumed to be allocated in that memory partition, provided that they are not allocated specifically to another one.

2.5.2 SW Platform View

The *SWPlatformView* defines the operating systems that are in the HW/SW platform. The operating systems are modelled by a UML component specified by the stereotype <<OS>>. The attributes associated with this stereotype are:

<<OS>>
type:String [1]
scheduler: Scheduler[*]
drivers: DeviceBroker [*]
interProcessCommunication: InterProcessCommunicationMechanism [1]

Table 4:Figure 58 OS stereotype attributes.

The type of the OS is defined in the *type* attribute (linux, windows, etc.).

The attribute *scheduler* defines the schedulers associated to the OS. The schedulers are modelled by the MARTE stereotype <<Scheduler>>. In this component, the scheduling policy can be annotated. The scheduling policy is captured in the attributes *schedPolicy* and *otherSchedPolicy*.

The attribute *schedPolicy* is an enumeration. The possible values considered in this methodology are “EarliestDeadlineFirst”, “FixedPriority”, “RoundRobin”… “Other”. In the case the value is “Other”, the scheduling policy is annotated in the attribute *otherSchedPolicy*.

The *driver* attribute of the stereotype *OS* enables association of *DeviceBroker*s with the OS component

The *interProcessCommunication* attribute defines the OS services that automatically create the communication infrastructure in order to communicate processes in the OS. Thus, code will be

created ad-hoc depending on which mechanism is specified for each OS instance. Five types of inter process communication mechanism are currently supported for automatic code generation. These types are:

- FIFO channels
- Sockets
- message queues
- shared memories
- files

Using this option, designers can easily explore the performance impact that each one has on the final implementation and select the most suitable ones for each system.

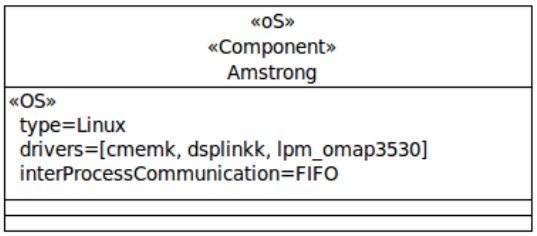


Figure 51 OS component.

Drivers

The *OS* components can have an associated set of drivers to provide access to peripherals or to manage specific processing HW resources of the platform. Drivers are modelled by the MARTE stereotype <<DeviceBroker>> applied on an UML component.

A *DeviceBroker* driver can have associated properties that enable well-defined driver specification:

- Repository: denotes the address where the driver can be downloaded,
- Parameter: denotes configuration information for the driver,
- Device: is the file for the control of the HW resource.

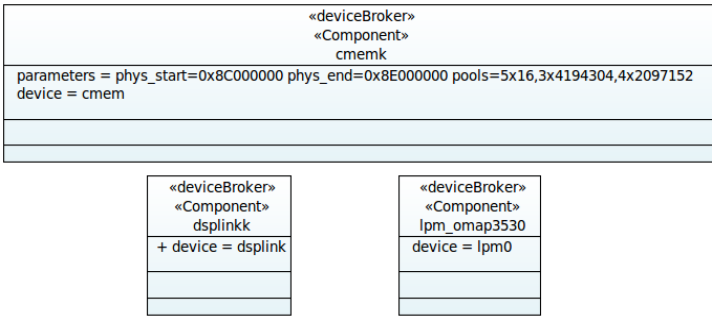


Figure 52 Driver for DSP management.

Repository

The "repository" property denotes the URL address of the repository where the driver can be downloaded in order to be installed in an automatic way. This property is captured in a UML property included in the *DeviceBroker* component. The name of this UML property should be "repository". The address is annotated in the attribute "Default Value" of the UML property, by using a UML Literal String attached to the "Default Value" attribute.

Parameters

The "parameters" property denotes the set of parameters required for a correct configuration of a driver. This property is captured in a UML property included in the *DeviceBroker* component. The name of this UMl property should be "parameters". Then, the set of parameters are annotated in an attribute "Default Value" of the UML property, a UML Literal String attached to the "Default Value" attribute.

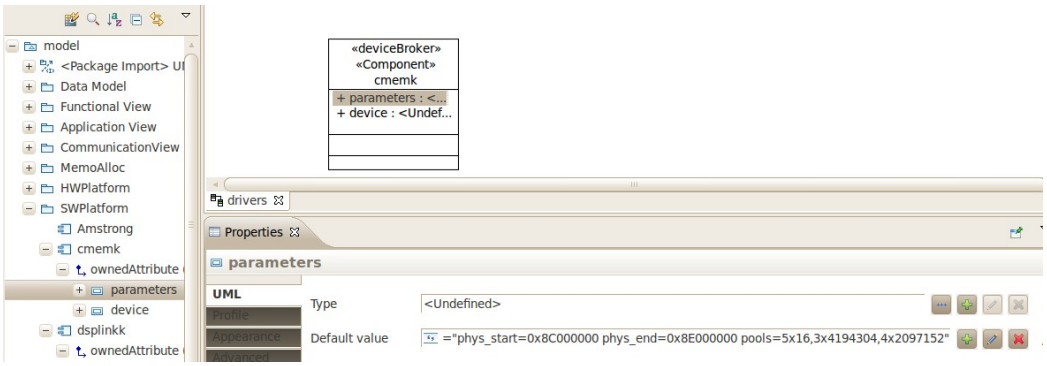


Figure 53 Parameter driver property.

Device

The "device" property denotes the device property required for a correct configuration of a driver. This property is captured in a UML property included in the *DeviceBroker* component. The name of this UMl property should be "device". Then, the set of parameters are annotated in an attribute "Default Value" of the UML property, a UML Literal String attached to the "Default Value" attribute.

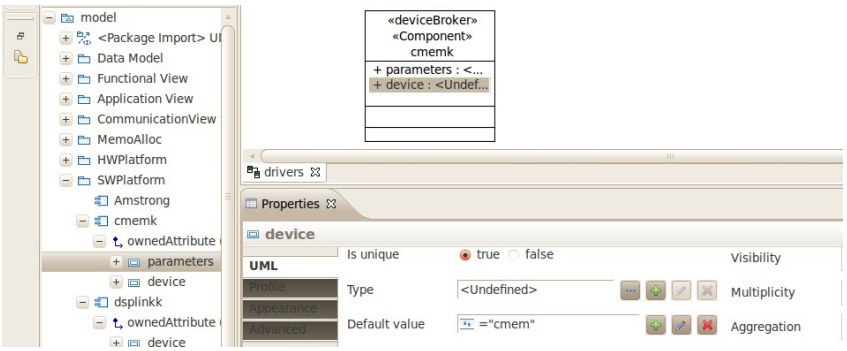


Figure 54 Device driver property.

2.5.3 HW Resources View

The *HwResourceView* declares all the HW components required for the specification of the platform architecture. In the *ArchitecturalView*, instances of the HW components declared in the HW Resources view will be used in the capture of the HW architecture.

The UML elements used in this view are the following:

- UML Components for modeling the HW component types,
- Class diagrams are used for defining the HW components,
- HW platform architecture, which includes a hierarchical partitioning of the complete HW architecture of the system in a single or various components, usually modeled using composite structure diagrams, including:
    - Instances of HW resources (processors, memories, buses, network, etc.),
    - Interconnections among these HW resources.

The MARTE stereotypes used to specify the HW components that can be captured in the *HwResourcesView* are shown in Table 5:

UML2 Diagram elements	MARTE profiles	MARTE stereotypes
Component	HRM	HwProcessor HwRAM HwROM HwCache HwBus HwMedia HwEndPoint HwBridge HwI_O HwISA

Table 5: MARTE stereotypes used for refining the HW platform.

Figure 55 shows a CPU with one ARM9 processor, data and instruction caches, a bus, main memory and an I/O device:

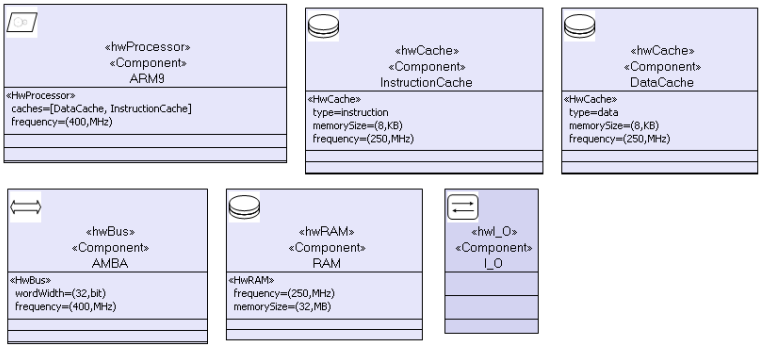


Figure 55 HW platform resources.

Physical Magnitudes

HW component attributes can be annotated with values, which can be either a-dimensional or represent a physical magnitude. The value of a physical magnitude is annotated in the following way:

- **ValueUnit**
    a. 100Mbps
    b. 2KB
    c. 10mW

The accepted units for each attribute and the default physical magnitude are shown in Table 6:

Attribute	Physical magnitude
frequency	GHz MHz KHz Hz
memorySize	TB GB MB KB B
wordWidth	b
BandWidth	Gbps Mbps Kbps Bps
memoryLatency	Us Ns
power	W mW uW nW pW
energy	J mJ uJ nJ pJ
blockSize	B Words

Table 6: HW attributes and physical units.

HW Processors

HW processors are modelled as components decorated with the MARTE stereotype <<HwProcessor>>.

Frequency

The frequency of the processors is captured in the *HwProcessor* attribute $frequency$.

Number of Cores

The number of cores that a processor has is defined in the *HwProcessor* attribute $nbCores$.

Speed Factor

A speed factor can be associated to a processor in the *HwProcessor* attribute $speedFactor$.

TDMA Slots

The *HWProcessor* may have associated the number of slots when it is directly connected to a TDM (in this case, the HW processor is assumed to have the network interface capabilities). This property is modelled as the attribute *assignedSlots: NFP_Integer*. Then, the value is annotated in the property "Default Value".

Cache

Each HW processor could have data and instruction cache memories. Thus, each HW processor can have associated a set of HwCaches instances. The caches can be associated to an *HwProcessor* by means of the attribute caches of the stereotype *HwProcessor* (Figure 56). This stereotype attribute selects the UML components that are characterized by *HwCaches*.

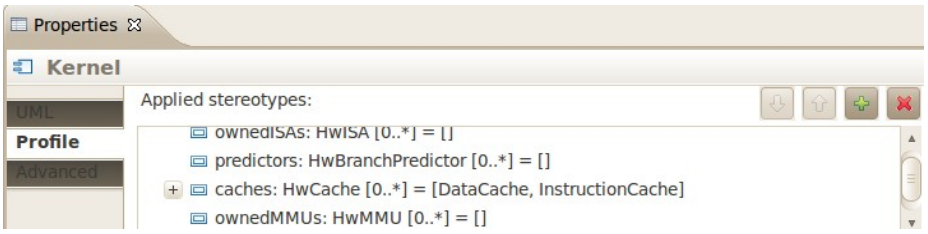


Figure 56 Associating caches to an HWProcessor.

Processor ISA

The *HwProcessor* can be more specifically defined by an Instruction Set Architecture (ISA). The MARTE stereotype <<HwISA>> is applied to a new UML component. This *HwISA* component is associated with the *HwProcessor* through the *HwProcessor* attribute *ownedISAs*. Two attributes of the *HwISA* stereotype are considered in this methodology:

- family: NFP_String. Defines the ISA family type,
- ISA_Type. Specifies the ISA type.

Currently, the possible values of the family attribute are:

- DSP,
- GPU,
- CortexA9,

- undef.

The Isa_type includes:

- RISC: Reduced Instruction Set Computer,
- CISC: Complex Instruction Set Computer,
- VLIW: Very Long Instruction Word,
- SIMD Single Instruction Multiple Data,
- Other,
- Undef.

Processor Caches

Cache memories are modelled by the MARTE stereotype *HwCache*. So, Table 7: shows the possible values of the *type* and *level* attributes of the *HwCache* stereotype that determine the type of cache.

HwCache attribute	Type of Cache
level = 1 & type = data	Data cache
level = 1 & type = instruction	Instruction Cache
level !=1 & type = unified	Unified Cache for caches of level more than one

Table 7: HwCache attribute values.

Figure 57 shows an example of caches components:

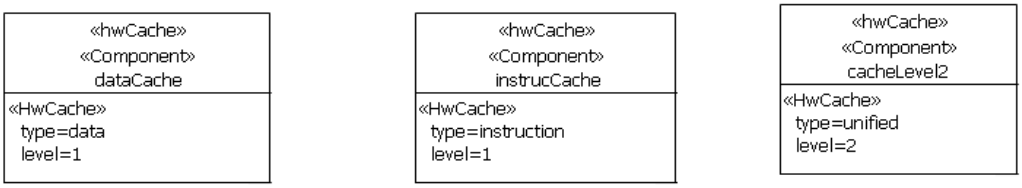


Figure 57 Cache components.

Additionally, the caches can be characterized with three additional attributes: the block size (specifies the width of a cache block), the associativity and the number of sets. These caches attributes can be specified in the attribute *structure* of the MARTE stereotype *HwCache*. The attribute *structure* is typed as *CacheStructure*:

HwCache attribute	Attributes
structure	blockSize associativity

Table 8: Definition of the structure attribute.

The specification of these attributes must be annotated as a string. The attributes annotation is shown in Figure 54. The attributes are identified as *"$BlockSize"* and *"$Associativity"*. Both data annotations are specified separated by semicolon. If *blockSize* has no specified unit, its value is interpreted in *Words.* Else, *blockSize* must be annotated in B (Bytes), as shown in**¡Error! No se encuentra el origen de la referencia.**:

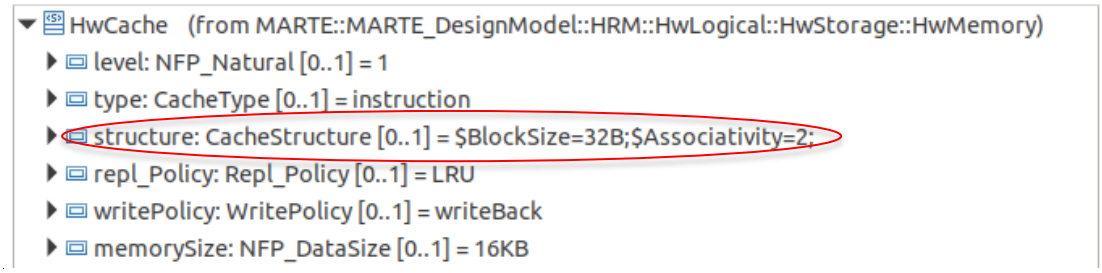


Figure 58 Specification of the cache attributes blockSize and associativity.

The word size associated to the cache memory is annotated in a UML property named *elementSize* of the *HwCache* component (Figure 59). When this attribute is not present, the default value annotated is 4 Bytes. The size of the caches is defined in the attribute *memorySize*. The type of write policy is specified in the attribute *writePolicy*. It can be *writeBack* or *writeThrough*. In the case the cache is typed as instruction (attribute type), another attribute can be captured; the size of the address. This property is annotated in the *HwCache* attribute *addressSize*.

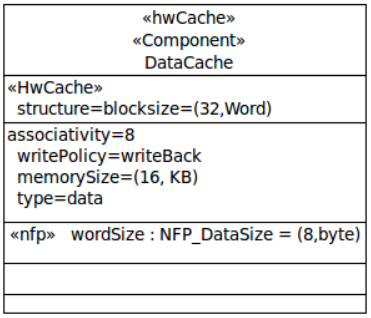


Figure 59 Cache specification.

Buses

Buses are modelled by the MARTE stereotype <<HwBus>>. Different properties characterize a bus.

Word width

The property word width specifies the word width per transaction expressed in bits or bytes and it is captured in the *HwBus* attribute *wordWidth.* It is expressed in bytes or bits. The default value of *wordWidth* is 8 bytes

Bandwidth

The property bandwidth specifies the number of transactions per second. It is captured in the HwBus attribute *bandwidth*. It is expressed in bits/s, Kbits/s, Mbits/s… The default value of the *bandWidth* is 1 Gbit/s.

Burst size

The property burst size denotes the number of event occurrences within a burst. It is specified in the *"blockT"* attribute of the *HwBus* profile and it is defined as *"$BurstSize=ValueUnit"*, with *unit* in Bytes. When this attribute is not present, the default value annotated is 8 words.

TDMA bus

For charactering a bus TDMA a set of specific properties should be captured. These properties are captured as UML attributes of a *HwBus* component. These attributes are the following:

- numberSlots: NFP_Integer
- timeSlot: NFP_Duration
- capacitySlot: NFP_DataSize
- payloadSlot : NFP_DataSize
- payloadRateSlot : NFP_DataTxRate
- timeCycle: NFP_Duration

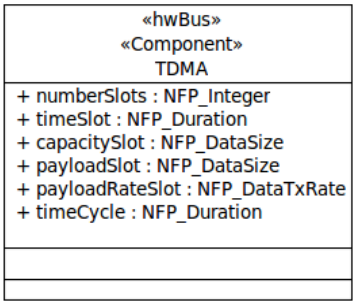


Figure 60 TDM bus component properties.

Then, in the property "Default Value" of each of the previous attributes, the individual value is annotated.

Bridges

In order to connect busses, bridge components should be used. These elements are modelled by the MARTE stereotype <<HwBridge>>. HwBridges only can connect HwBus component. The only property considered is the frequency.

Memories

The memories are modelled by the MARTE stereotypes <<HwRAM>>, <<HwROM>> or <<HwMemory>> according to the type of memory to considerer.

Memory size

The size of the memory is annotated in the attribute *memorySize*.

Memory latency

The memory latency attribute is annotated as a comment on the memory component as *"$Latency=ValueUnit"*.

Networks

As commented above, nodes in a network are stereotyped with <<ComputingResource>>. Nodes communicate each other through ports stereotyped as <<CommunicationEndPoint>>. Ports are linked by edges stereotyped as <<CommunicationMedia>>. The properties of the link will depend on the kind of network used (i.e. Ethernet, internet, Wi-Fi, etc.). Network hierarchy is supported in the same way as functional hierarchy.

I/O Components

The MARTE stereotype <<HwI_O>> models the HW component used as I/O system device.

HW components Functional Modes

HW components can have different associated functional modes that specify different characteristics that define the HW component's behavior according to a set of configuration parameters. These functional modes are defined by attributes: *frequency*, *voltage*, *dynamic power* and *average leakage*. In addition, the transitions among the functional modes are characterized as well. The transitions among modes are characterized by the time consumption in the mode transition and the power consumption in the mode transition.

In order to model these functional modes, the corresponding HW component should have a UML state machine. In a UML state diagram, the HW component modes and the mode transitions are captured. The HW component modes are represented as UML states specified by the MARTE stereotype <<Mode>>. The mode transitions are represented as UML transitions specified by the MARTE stereotype <<ModeTransition>>.

In order to characterize the functional attributes previously mentioned, some modelling elements have been used. The first one is taken from [ASH12], specifically the stereotype <<HwPowerState>>, in order to specify the *frequency* of the HW component in this mode. The attribute *Pstatic* of the *HwPowerState* enables to capture the power consumption in idle in this mode. The dynamic power of the mode is defined by the application of the MARTE stereotype <<ResourceUsage>>, specifying the attribute *powerPeak*. In order to define the last two attributes of a functional mode, *voltage* and *average leakage,* two UML comments should be associated with the corresponding UML state. There, both values are annotated. All the attribute values should be annotated as the MARTE specifies in order to define the non-functional properties (value, unit).

In order to characterize the mode transitions, the power and the time consumption have to be defined. The time consumption is defined in the attribute *setupTime* owned by the stereotype *HwPowerStateTransition* defined in the previously mentioned paper. The power consumption is specified by the stereotype <<ResourceUsage>>.

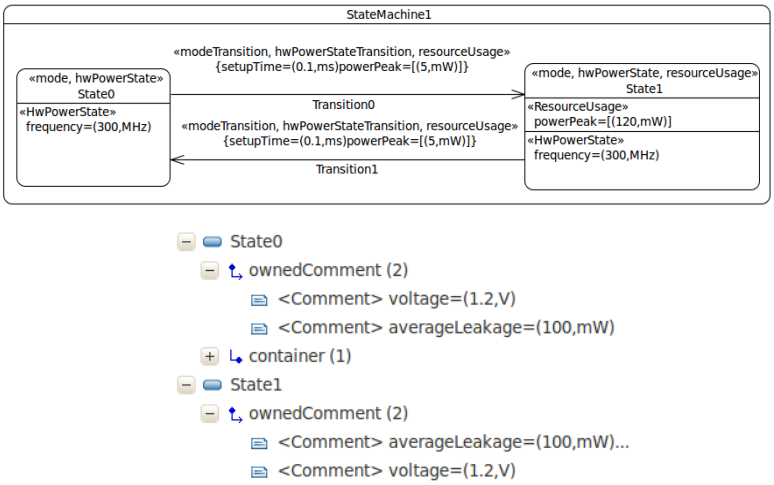


Figure 61 HwProcessor mode specification.

Power Consumption

HW components have associated static and dynamic power consumption. This value is annotated to the HW component as a comment in the following fashion:

- Static Power: *"$StaticPower=ValueUnit",* with Unit in Watts (uW, mW…)
- Dynamic Power: *"$Dynamic Power=ValueUnit",* with Unit in Joules (pJ, nJ…), or in Amperes by Hertz. If defined as the latter, another comment specifying the supply voltage of the component should be specified as *"$Voltage=ValueUnit"*, with unit in Volts.

Furthermore, caches have associated two dynamic energy consumptions: the consumption of a hit and the consumption of a miss. They are captured by adding a comment to the related cache as *"$HitEnergy"* and *"$MissEnergy"*, using the same units as for the general dynamic power.

Files

In the same way as the files that store the implementation source-code of the applications are modeled as UML artifacts, so the files describing the HW components. These artifacts are specified by the UML standard stereotype <<File>>. The Artifacts are specified by a name (annotated in the attribute "name") and in the attribute "File name", where the name and the extension of the file should be included.

2.5.4 HW implementation view

Among the HRM stereotypes, there are three which correspond actual with a kind of implementation and not to a proper HW resource. They are the following:

- HwComponent. In S3D, the HwComponent class is associated to a commercial-off-the-self component, usually an integrated circuit. When a HW platform object is mapped to a HwComponent is because this object is part of the architecture of the commercial device,
- HwPLD. In S3D, the HwPLD property is associated to an FPGA implementation. In that case, the model could be used to automatically generate the HDL description (VHDL or Verilog) to be synthesized on the FPGA,

- HwASIC. In S3D, the HwASIC property is associated to an application-specific implementation of the platform object in an integrated circuit. In that case, the model could be used to automatically generate the HDL description (VHDL or Verilog) to be synthesized on the ASIC.

2.6 PSM View

2.6.1 Architectural View

The Architectural view captures the platform specific model (PSM) as a mapping of the PIM onto the PDM. The platform specific model is captured as a single or several components containing the following items:

- Mapping of architectural components to memory partitions or directly to OSs in case only an executable is going to be generated on this OS,
- Mapping of memory partitions to OSs or directly to processors in case a bare-metal executable is going to be generated on this processor,
- Mapping of OSs to processors.

The *Architectural View* contains the System component, i.e. a component decorated by the <<System>> stereotype. The System component of the architectural view represents the platform specific model. Composite structure diagrams are associated to the system component, and used to capture the HW/SW architecture of the platform as it was shown in Figure 10 and Figure 11. Mapping is made by means of UML abstractions decorated with the MARTE <<allocate>> stereotype.

Interfaces between the system and the environment are mapped to the high-level driver. It will make use of the low-level driver of the HW peripheral in the corresponding OS where the interface is going to be called. This driver should be provided apart. In S3D this information is taken from the IP-XACT [IPXA14] description of the device.

Concrete information out the physical implementation of the HW resources can be modeled by using the three kind of implementation alternatives considered, an FPGA (<<HWPLD>>), an ASIC (<<HWASIC>>) or a COTS (<<HWComponent>>).

An application component can be mapped directly to an implementation alternative as shown in Figure 62. This should be interpreted as a direct implementation in HW of the corresponding functionality. This implies RTL synthesis, if the VHDL code for the application component is provided, or behavioral synthesis if the C/C++ code is provided and it satisfies the additional requirements imposed by the behavioral synthesis tool.

2.7 Verification View

The Verification View defines the structure of the system environment. The environment has to be thoroughly defined in order to enable the execution of the performance estimation tools during the design process with appropriate inputs.

The environment structure consists of environment components that interact with the system. Additionally, these environment components have the associated functional elements that define their functionality. The modeling of the environment makes use of a set of stereotypes of the UML standard profile UTP.

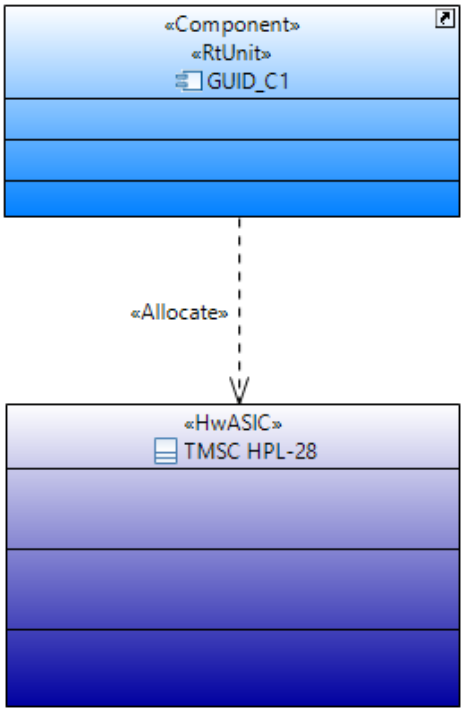


Figure 62 Application mapping for HW synthesis.

2.7.1 Environment components

The environment components represent the devices that interact with the System. The environment components are modelled as UML components. This set of UML components is specified by means of stereotypes included in the standard UML Testing Profile (UTP). The components that compose the system environment are defined in a UML class diagram. These components are specified by the UTP stereotype <<TestComponent>>, as shown in Figure 67:

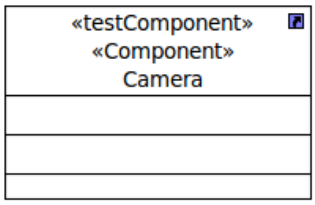


Figure 63 Environment component.

Environment component Functionality

Each environment component has an associated specific functionality. This functionality is modelled as UML components specified by the MARTE stereotype <<RtUnit>> and the UTP stereotype <<TestComponent>>, as shown in Figure 64. The environment application components should be included in the *ApplicationView* like the rest of the application components of the system.

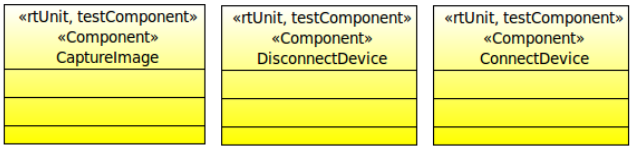


Figure 64 Environment application components.

All these RtUnit-TestComponent components can have the same associated modeling elements (threads, file folder, files) as the application components. Each application component has associated C/C++ files. These C/C++ files are file artifacts defined in the Functional View. The files should be specified by the UML standard stereotype <<File>> and the stereotype <<ApplicationFile>>. The files used for defining the functionality of the environment should be typed as *environment=true*. The assignation of the file artifacts is done through a UML abstraction specified by the MARTE stereotype <<allocated>> (Figure 65).

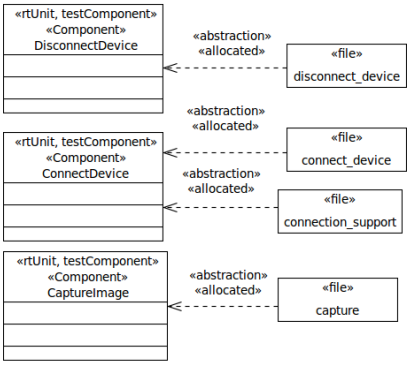


Figure 65 Environment Application components with associated Files.

Environment component structure

Each environment *TestComponent* component has internal parts that are the environment application components. The internal functional structure of the environment *TestComponent* component is captured by using instances of *RtUnit-TestComponent* application components (Figure 66) in a Composite structure diagram associated with the environment *TestComponent* component.

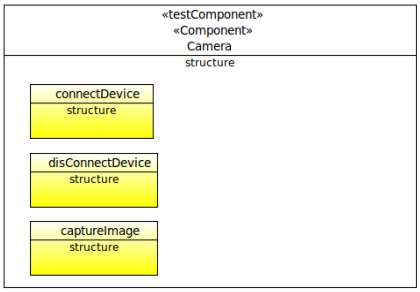


Figure 66 Application instances of an environment component.

Environment component structure: ports

The communication is established through ports. The ports specify the required/provided interfaces by means of which the components interact among them. The ports are specified by the MARTE stereotype, being defined as *provided* or *required*, where an interface is associated.

The ports that have been specified by the *ClientServerPort* stereotype are those of the environment component (*TestComponent*), as can be seen in Figure 67 (Camera *TestComponent*). These *TestComponent* ports are connected to the internal application instance ports by using UML connectors (Figure 71). These application instance ports have to be named similarly to the *TestComponent* port that they are connected to (Figure 67).

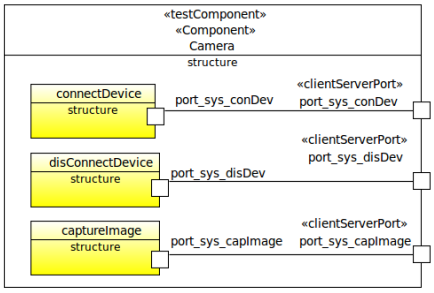


Figure 67 Environment Application components.

2.7.2 Environment structure

The environment structure is composed of instances of environment components connected to the *System*. The environment structure is modelled in a UML component specified by the UTP stereotype <<TestContext>>. The environment structure is modelled in a UML composite structure diagram associated with this *TestContext* component. This composite structure diagram contains instances of *TestComponents* and a property typed by a *System* component; specifically, the *System* component defined in the *Application View* since the port that interacts with the environment is defined in this *System* component included in this model view; this *System* property should be specified by the UTP stereotype *SUT* (System Under Test).

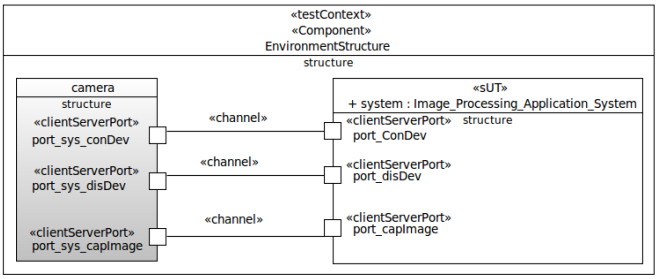


Figure 68 Definition of the environment structure

Then, in order to define the semantics of channels among the *TestComponents* and the *System,* UML connectors should be specified by the stereotype *Channel*, specifying the type of communication media defined in the *CommunicationView*.

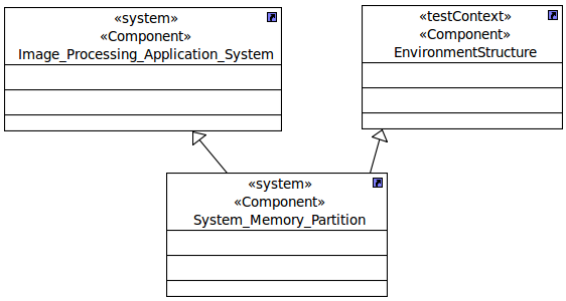


Figure 69 Generalization of Environment structure with the System component of the MemorySpaceView.

2.7.3 Memory allocation

The Environment elements must be allocated to memory spaces. The *TestContext* component has to be associated with the *System* of the *MemorySpaceView*. This *System* component should be specialized by the *TestContext* component defined in the *VerificationView*. This specialization is modeled by means of a UML generalization defined in a UML class diagram (Figure 69).

Then, the allocation on memory spaces of the environment component (instances of *TestComponent* components) can be done (Figure 70).

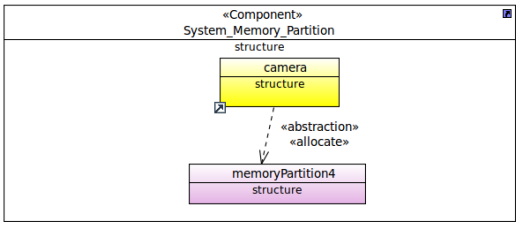


Figure 70 Allocation of environment component to the memory partitions.

This view is not mandatory. The reason is that the methodology considers an alternative solution. As described above, different files can be associated with the system. Using this feature, systems with minimal environments can be modelled directly indicating the source file with the environment code instead of creating a complete specific view.

2.7.4 Modelling Data Dependencies

S3D supports data dependencies analysis in order to verify whether timing constraints are fulfilled or not. In S3D methodology, data dependencies and data paths are represented with UML sequence diagrams. For this analysis S3D uses traces generated during the execution of the application, which are generated using the *Common Trace Format* [CTF13], a standardized binary trace format designed for a fast and efficient writing while using few disk space.

Data dependencies are represented using UML Sequence diagrams and included inside the verification view.

Different parts (i.e. component instance, that is, *property*) are represented with *lifelines*, one per part involved in the data path to be analyzed. Lifelines are related to their corresponding properties through the "*Represents"* box, as can be seen in Figure 71.
An execution of a service or function is represented over the lifeline with an *Action Execution Specification.* Since a specific part (component) can only have one main function, i*f* the Action Execution is not pointed by a service call, it will unequivocally represent the main function of the component. On the contrary, if the Action Execution is directly pointed by a service call, it represents that specific service in question. For periodic functions, each Action Execution represents one iteration of the total run of the service.
Service calls are represented using *Message Sync/Async.* Synchronous messages are represented with filled arrows, while asynchronous messages are depicted with empty arrows. Both can be used, since the synchronicity does not depend on the type of the message, but on the *operation* properties. Synchronous messages only allow adding a temporal restriction about when the service is requested

from the client, while asynchronous messages also enable specifying when the server executes the service.

Furthermore, is important to consider that synchronous messages coming from an Action Execution can only point to another Action Execution, whereas asynchronous messages can point directly to the lifeline. Thus, a call pointing to the lifeline and one pointing to an Action Execution which finishes without producing any call are identical in terms of data path analysis, and if not required, the omission of irrelevant Action Execution boxes is recommended for diagram simplicity. Let us use the following diagram to illustrate this.

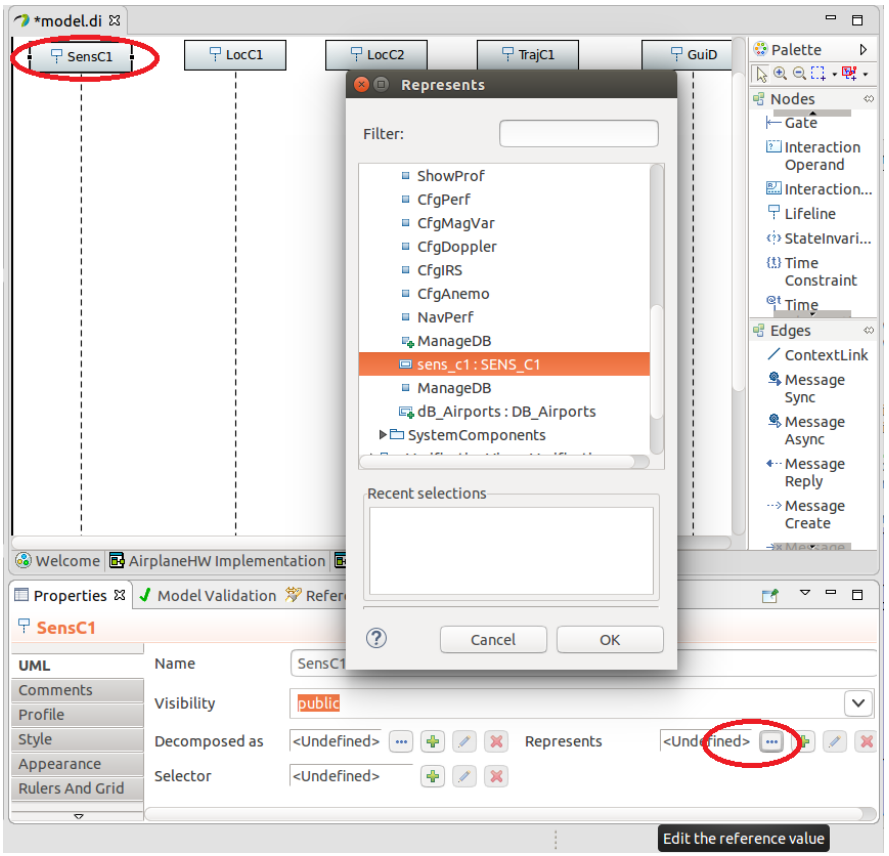


Figure 71 Linkage of lifelines to parts

As shown in Figure 72, first Action Execution represents the run of the main function of *sens_c1* component. This function makes a synchronous call to service *trSensorData* from *loc_c1* component, passing the resulting data to the main function of *loc_c1*. This main function makes an asynchronous service call to *trHighFreqBCP*, which passes resulting data to the main function of *loc_c2.* Both *trSensorData* and *trHighFreqBCP* are executed, but since *trHighFreqBCP* is executed from an asynchronous call, its Action Execution box can be omitted, simplifying the diagram. Therefore, if possible, asynchronous calls are recommended.

When placing the messages from the client to the server lifelines, a pop-up will appear asking for the operation requested. If the *operation* has already been specified in the *Component (*or its *generalizations,* or the *generalizations* of its *generalizations*, i.e. *components* or *interfaces*), it will appear in the list.

Otherwise, a new *operation* should be created, placing the name of the *operation* as shown in Figure 74. Note that, in that case, you also have to navigate through the model to edit the new *operation* just created, in order to add its details (i.e. function arguments).

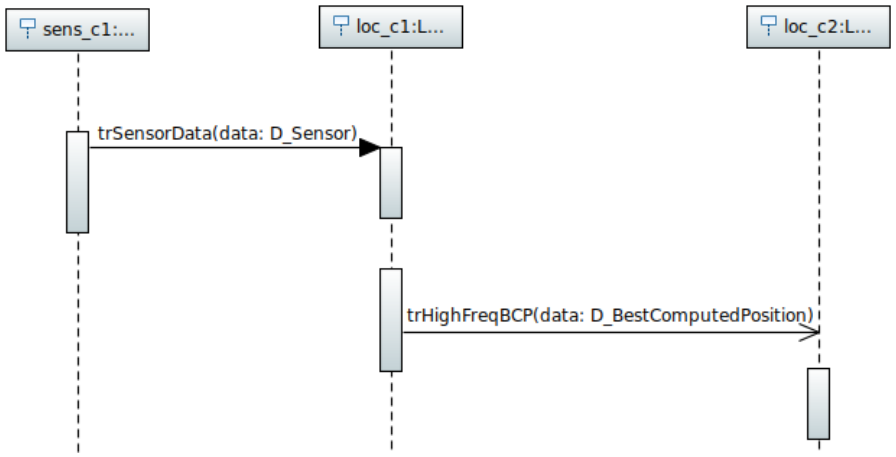


Figure 72 Action Execution Specification usage

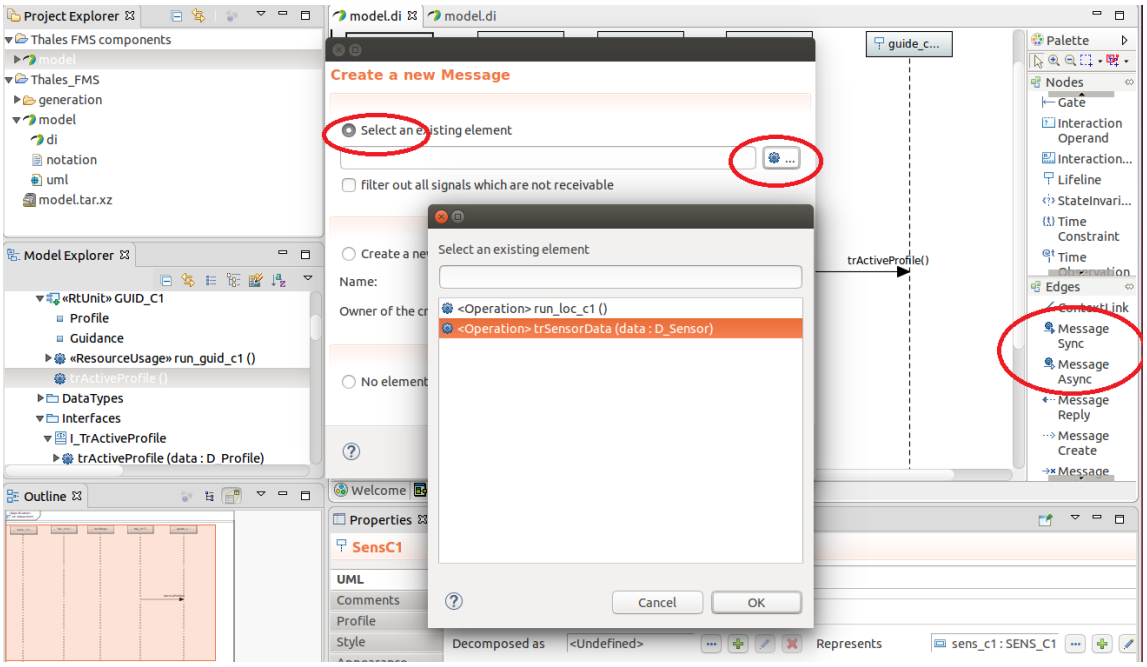


Figure 73 Existing message operation assignment

Once all service calls have been placed in the diagram, a *Time Constraint* must be added at the beginning of the last *Message* of the chain, and maximum and minimum time values are specified as described in Figure 75 (use a *LiteralString* value, as shown).

It is important to notice that the Y coordinate of the diagram (vertical) corresponds with time (or causality). The order of the calls in the diagram must correspond with its requested execution during real operation: if call to A must happen before call to B, A *must* be higher than B in the diagram.

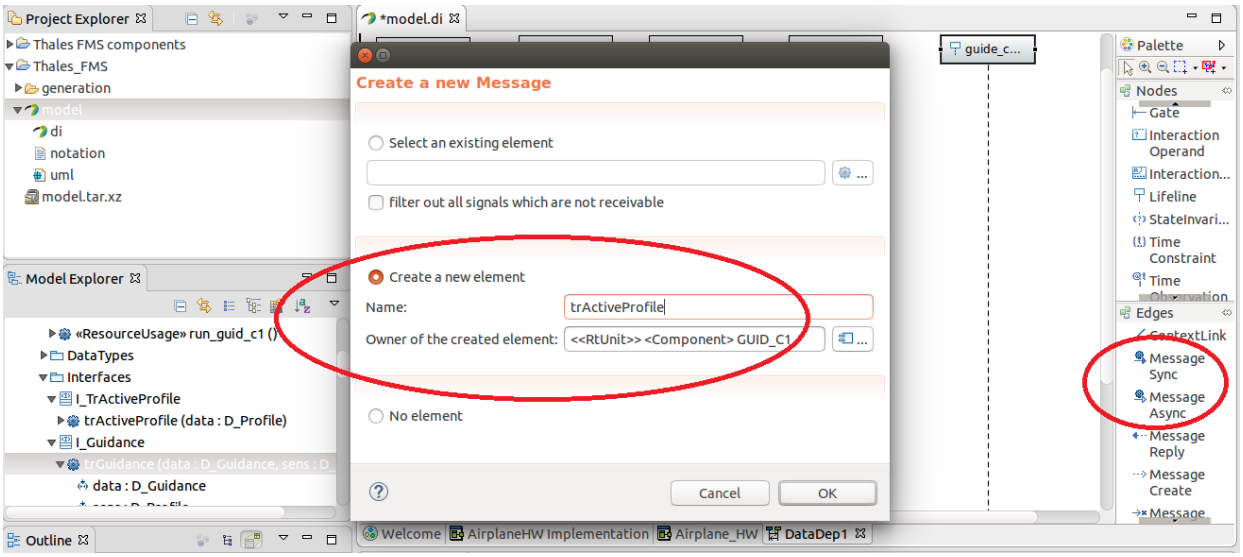


Figure 74 Creating new message operation assignment

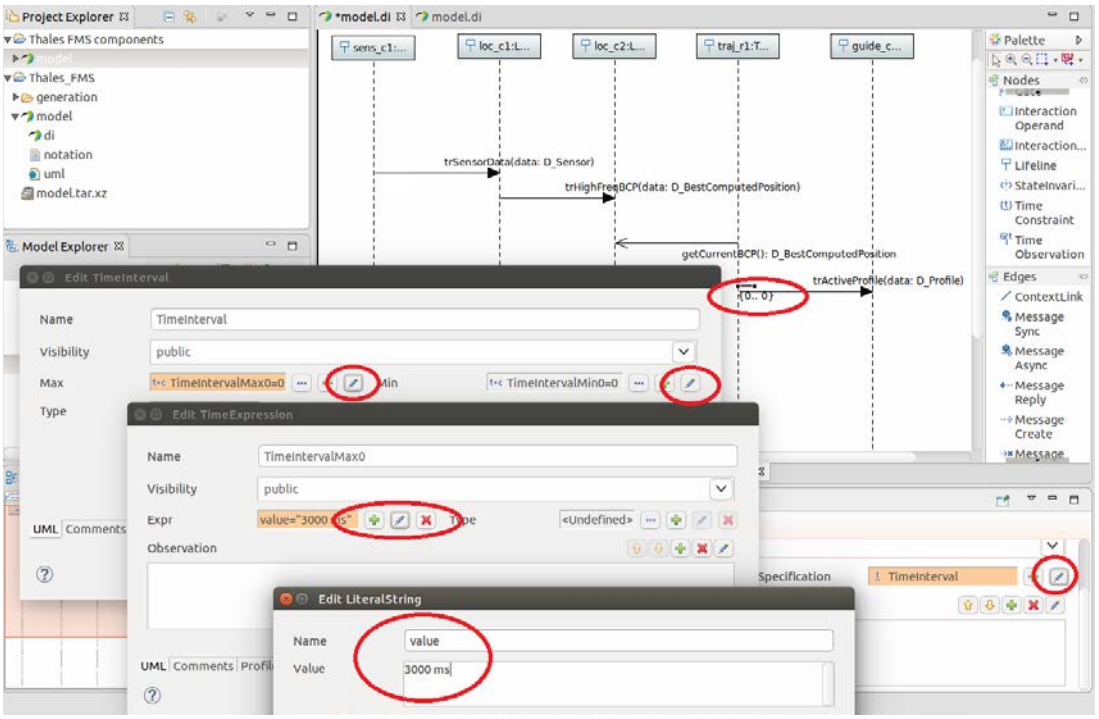


Figure 75 Time constraint specification in the sequence diagram

3 S3D System Modeling under different MoCs

In this document, the modeling of interfaces among functional components in the S3D framework is described. The goal is to provide the system engineer with a flexible modeling infrastructure able to support different system engineering methodologies. 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 modeling paradigms like Object-Oriented Modeling or Actor-Oriented modeling in a uniform and unified way.

3.1 Object-Oriented Modeling

In Object-Oriented Modeling (OOM[2]), 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 waiting for the returning data, 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 problem with this kind of SW modeling have been highlighted many times [Lee06].

Nevertheless, S3D 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 released. A CBSM methodology such as S3D, 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.

3.2 Actor-Oriented Modeling

In Actor-Oriented Modeling (AOM[3]), the system is conceived as a collection of concurrent components called actors. Actors encapsulate data and functionality and interact each other through predefined communication patterns, which may lead to concrete Models of Computation [LeNe04]. 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 and languages are Simulink[4], Labview[5], Modelica[6], VHDL [TTOV97], Verilog[7], SystemC[8], and Ptolemy[9].

3.3 Interface modeling

In this document, we will show that the S3D CBSM can model systems in a general and unified way able to support both OOM and AOM under different Models of Computation [LeSa96]. This is achieved by defining the properties of the functions in the provided/required interfaces. 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:

InterfaceFunction(X1, … Xn, Z1, … Zm);

---

[2] https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design
[3] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.230.1295&rep=rep1&type=pdf
[4] https://es.mathworks.com/products/simulink.html
[5] http://www.ni.com/en-us/shop/labview.html
[6] https://openmodelica.org/
[7] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1620780
[8] http://accellera.org/downloads/standards/systemc
[9] http://ptolemy.eecs.berkeley.edu/ptolemyII/

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:

$Xi \equiv Zj$

3.3.1 Properties of the services of the interface

Each interface function 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].*

The 'ConcurrencyKind' enumeration has three possible values:

reader. The execution of the service has no side effects. 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).

writer. The execution of the service may have side effects. Consequently, once the service is provided any call to any other service should be blocked.

parallel. The service can be provided concurrently (with the concurrency limit defined by the **srPoolSize** attribute of the corresponding component).

*The enumeration 'CallConcurrencyKind' of the '**concurrency**' property [1] = sequential.*

Any MARTE RtService is an UML 'BehavioralFeature' and, as such, inherits the enumeration 'CallConcurrencyKind' of the attribute 'concurrency':

sequential. No concurrency management mechanism is associated with the BehavioralFeature and, therefore, 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. 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. Multiple invocations of a 'BehavioralFeature' that overlap in time may occur to one instance and all of them may proceed concurrently.

The following table resumes the behavioral interpretation for all the combination possibilities of the two attributes. As a result, regarding how many services can be attended in parallel in the port, the 12 possibilities can be reduced to only two:

G: Only one call to the service can be attended each time.

C: As many calls of the service can be attended in parallel as determined by the **srPoolSize** attribute of the corresponding component.

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 RtUnit.

		concPolicy: ConcurrencyKind [0..1]			
		reader	**writer**	**parallel**	**none**
concurency: CallConcurrencyKind [1..1]	**sequential**	**G**: Only one call to the service is attended each time. The service can be executed in parallel to other 'reader' services in the same interface or another in the same component	**G**: Only one call to the service is attended each time. The service cannot be executed in parallel to any other service in the same interface or another in the same component	**C**: Parallel calls to the service can be attended but the service cannot be executed in parallel to any other service in the same interface or another in the same component	**G**: Only one call to the service is attended each time. The service cannot be executed in parallel to any other service in the same interface or another in the same component
	guarded	**G**: Only one call to the service is attended each time. The service can be executed in parallel to other services in the same interface or another in the same component	**G**: Only one call to the service is attended each time. The service cannot be executed in parallel to any other service in the same interface or another in the same component	**G**: Only one call to the service is attended each time. The service can be executed in parallel to other services in the same interface or another in the same component	**G**: Only one call to the service is attended each time. The service cannot be executed in parallel to any other service in the same interface or another in the same component
	concurrent	**C**: Parallel calls to the service can be attended and the service can be executed in parallel to any other service in the same interface or another in the same component	**C**: Parallel calls to the service can be attended but the service cannot be executed in parallel to any other service in the same interface or another in the same component	**C**: Parallel calls to the service can be attended and the service can be executed in parallel to any other service in the same interface or another in the same component	**C**: Parallel calls to the service can be attended but the service cannot be executed in parallel to any other service in the same interface or another in the same component

Table 9: Interpretation of the combinations of concurrency-related attributes.

*The enumeration 'ExecutionKind' of the '**exeKind**' attribute [0..1].*

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 by the computing resource on which the called component has been mapped.

localImmediate. The execution is performed immediately by the computing resource on which the calling component has been mapped. This possibility is not yet considered.

*The Boolean attribute '**isAtomic**' [1] = false.*

When true, implies that the RtService executes as one indivisible unit, non-interleaved with other RtServices. This attribute does not affect the model of computation involved in the communication/synchronization mechanisms of the components.

*The enumeration 'SynchronizationKind' of the '**syncKind**' attribute [0..1].*

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.

3.3.2 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 not standard enumeration 'FullPoolPolicyKind' of the not standard '**fullPoolPolicy**' attribute [0..1].*

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 depending on the scheduling policy selected is removed and the new call stored.

removeLast. The last call to be attended depending on the scheduling policy selected 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.

3.3.3 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].*

The 'PoolMgtPolicyKind' enumeration has four 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.

*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.

3.4 Models of Computation

Depending on the properties defining the services and the provided and the required ports, different programming models corresponding to different Models of Computation (MoC) can be supported. Some fundamental MoCs require strict point to point synchronization/communication interfaces, that is, each interface involves a single provider component and a single required component. We will address them next. More complex communication interfaces will be addressed afterwards.

3.4.1 Point to point interfaces

Function Call/RPC/RMI

Under the Function Call (FC) MoC, the calling thread is stopped until the required function call is attended and the output data read as shown in Figure 76. Nevertheless, in order to avoid deadlocks, a timeout can be defined.

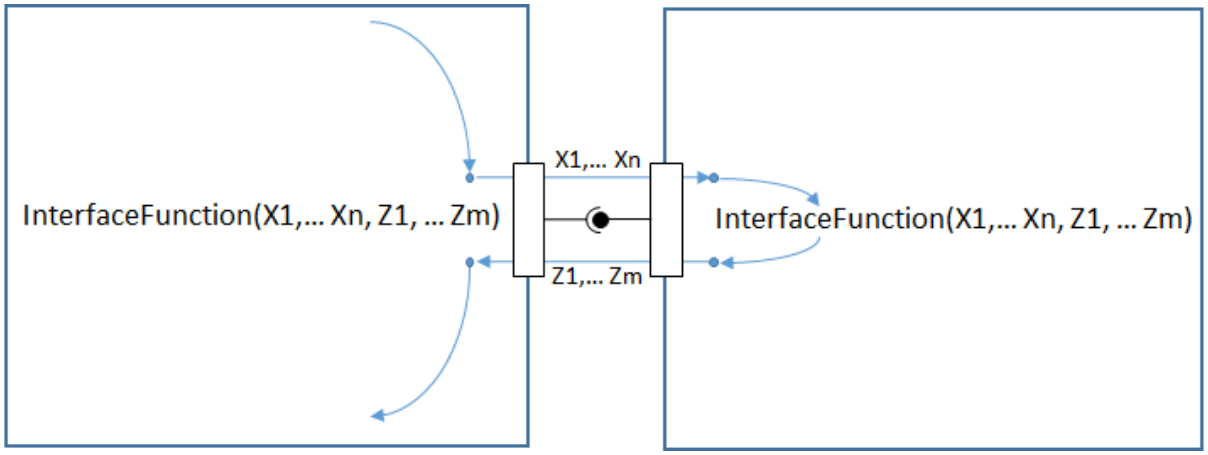


Figure 76 RPC MoC.

Table 10: 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.

Required Port		RtService			Provided Port		
NotAttendedService	retry	concurrency	exekind	syncKind	queueSize	FullPoolPolicy	MoC
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
dynamic	none	G or C	rem.Im.	sync.	none	none	exactly once

dynamic	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 10: RPC/RMI MoC.

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:

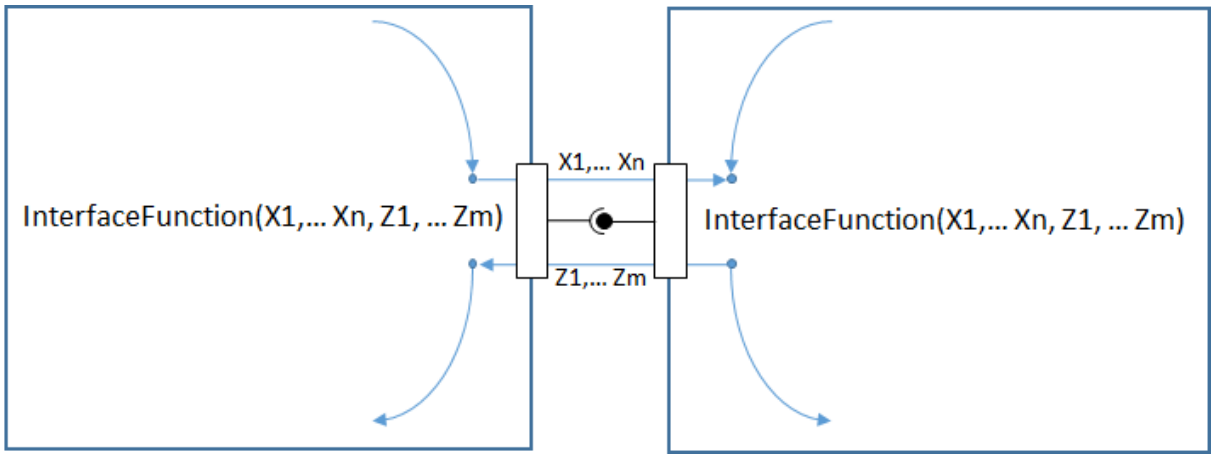


Figure 77 Rendezvous MoC.

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 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:

Required Port		RtService			Provided Port		
NotAttendedService	retry	concurrency	exekind	syncKind	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

Table 11: RV MoC.

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. 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 Khan 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.

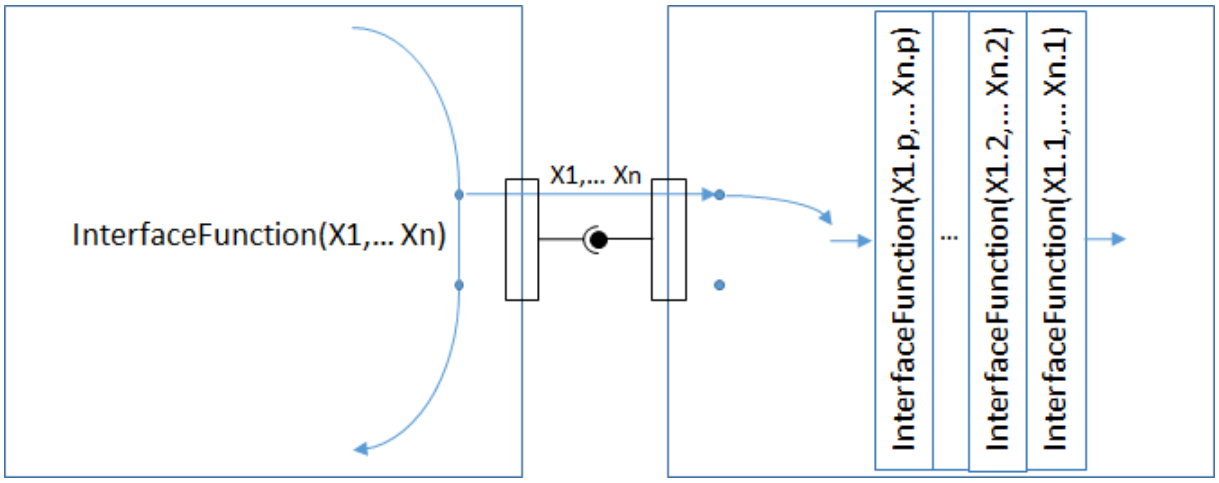


Figure 78 Data Flow MoC.

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 continues. In the KPN MoC, the 'NotAttendedService' attribute is 'infiniteWait'.

It is worth mentioning that when the execution time of the provided service is smaller than the rate at which it is called and the provided service changes the internal state of the component (i.e. a write function writing a data to be consumed by an internal thread in the component), the DF mechanism so defined does not ensure that a write-write race may occur. If this is the case, the providing RTService should be annotated as 'rendezvous'.

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. 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 [BELP96].

The maximum number of function calls to be stored is defined with the attribute 'queueSize'. 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:

Required Port		RtService			Provided Port		
NotAttendedService	retry	concurrency	exekind	syncKind	queueSize	FullPoolPolicy	MoC
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 12: DF MoC.

Discrete Event (DE), Time-Triggered (TT), Timed Data Flow (DTF)

In DE[10] systems, components react to events in their inputs. An event is an instantaneous indication to trigger a reaction. In S3D this can be modeled with an interface function without arguments, such as:

Notify();

In some cases, an event is a change in the value of a data. In some other cases, just the writing of a data with a new value even if it is the same as the previous one may be considered an event. It may happen that in case the receiving component it not available at the instant the event is notified, it get lost.

DE systems may be non-deterministic producing different results depending on which component is executed first when two simultaneous events occur triggering both. In order to avoid non-determinism in DE systems, a global control component can be used. The functional components divides its behavior in two phases. In the first one, the component reacts to the events in the inputs and compute results. In the second phase, it up-dates values in the outputs. The control component is in charge of synchronizing the evaluation-update phases of all the components.

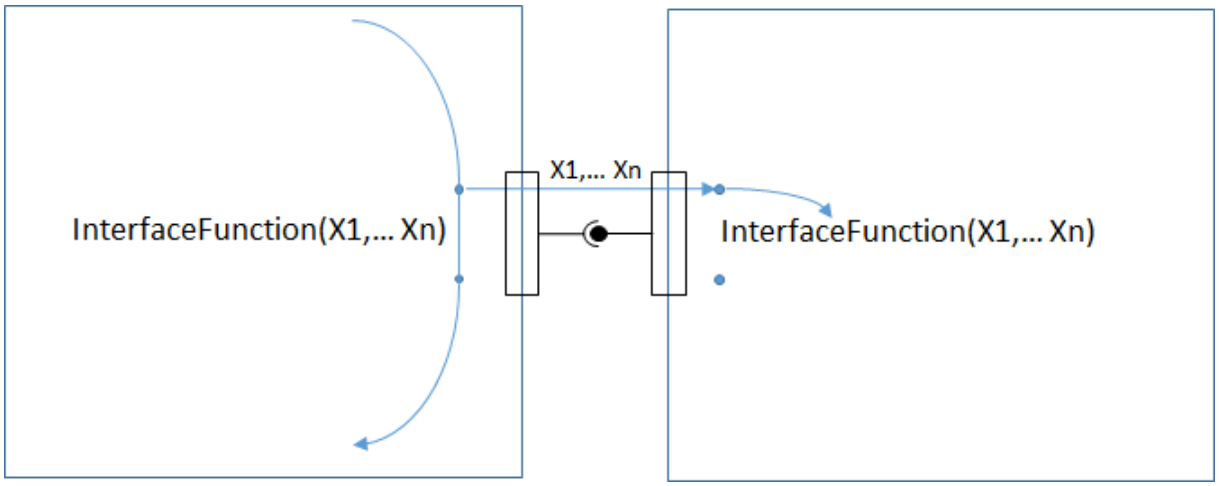
In the TT[11] MoC, the moment in which each component reads the inputs and the time in which it delivers the outputs are known in advance. In some cases, a clock is used in order to synchronize the input and output times of all the components.

The Timed-Data Flow (DTF) MoC is basically a TT MoC in which the frequency of each component may by different depending on the input and output rates. This is the MoC used by many analog simulators like Modelica, Simulink and SystemC-AMS. In order to accelerate simulation, the

[10] https://pdfs.semanticscholar.org/5a22/af628426a44c0bcbddd26b4c31c44a99af35.pdf
[11] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=739743

frequency of the underlying clock in each component can be decided dynamically, leading to the Dynamic Timed Data Flow (DTDF[12]) MoC.

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



These systems requires no blocking attributes at dispatching and returning and no buffering in the provided interface:

Required Port		RtService			Provided Port		
NotAttendedService	retry	concurrency	exekind	syncKind	queueSize	FullPoolPolicy	MoC
dynamic	none	G or C	rem.Im.	async.	none	none	DE/TT/TDF

Figure 79 Discrete-Event, Time-Triggered and Timed Data-Flow MoCs.

Synchronous Reactive (SR)

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

[12]www.accellera.org/images/resources/articles/amsdynamictdf/Whitepaper_SystemC_AMS_Dynamic_TDF_September_2011.pdf

4 References

[Amb15] S. Ambler: “Single source information: an Agile best practice for effective documentation”, 2015, http://agilemodeling.com/essays/singleSourceInformation.htm.

[ASH12] T. Arpinen, E. Salminen, T.D. Hämäläinen & M. Hänniikäinen. “MARTE profile extension for modeling dynamic power management of embedded systems”. JSA, April, 2012, pp.209–219.

[BELP96] Bilsen G., Engels M., R. Lauwereins R. and Peperstraete J.A. (1995) Cyclo-static data flow. International Conference on Acoustics, Speech, and Signal Processing. IEEE.

[BCW12] M. Brambilla, J. Cabot & M. Wimmer. “Model-Driven Software Engineering in Practice”, Morgan&Claypool, 2012.

[CTF13] Mathieu Desnoyers, EfficiOS Inc., Linux Foundation. “Common Trace Format (CTF) Specification (v1.8.2)”, 2013, https://diamon.org/ctf/

[GHH13] K. Grüttner, P.A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero, R. Valencia: "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration", Microprocessors and Microsystems, V.37, N.8-C, Elsevier, pp.966-80, 2013.

[HMV17] 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.

[HPP12] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero & R. Valencia: "A MDD Methodology for Specification of Embedded Systems and Automatic Generation of Fast Configurable and Executable Performance Models", ESWeek 2012 Compilation Proceedings, CoDes+ISSS’12, ACM, 2012.

[HPP14] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia & G. Palermo: "The COMPLEX methodology for UML/MARTE modeling and design-space exploration of embedded systems", Journal of Systems Architecture, V.60, N.1, Elsevier, pp.55–78, 2014.

[IPXA14] “IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows”, Accellera, 2014.

[KRB13] S. Korra, S.V. Raju and A.V. Babu: “Strategies for Designing and Building Reusable Software Components”, International Journal of Computer Science and Information Technologies, V.4, N.5, 2013.

[LaCo17] K-K. Lau & S. di Cola: “An Introduction to Component-Based Software Development”, World Scientific Publishing Company, 2017.

[Lap07] P. A. Laplante: “What Every Engineer Should Know about Software Engineering”, CRC Press, 2007.

[LaDi17] K-K. Lau & S. Di Cola S: “An Introduction to Component-based Software Development”, World Scientific Publishing, 2017.

[Lee06] E. Lee: “The Problem with Threads”, Computer, V.39, N.5, ACM, pp. 33-42, 2006.

[LeNe04] Lee E.A. and Neuendorffer S. (2004) Actor-Oriented Models for Codesign. In: Gupta R., Guernic P.L., Shukla S.K., Talpin JP. (eds) Formal Methods and Models for System Design. Springer.

[LeSa98] E. A. Lee and A. Sangiovanni-Vincentell: “A framework for comparing models of computation”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE, 1998.

[MMRT17] “Deliverable D1.1: Industry requirements specification”, MegaMart project, 2017.

[NMP17] E. di Nitto, P. Matthews, D. Petcu & A. Solberg (Eds.): “Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach”, Springer Open, 2017.

[PHV10] P. Peñil, F. Herrera & E. Villar: "Formal Foundations for MARTE-SystemC Interoperability", Forum on specification & Design Languages 2010, FDL'2010, IEEE, 2010.

[PNP14] H. Posadas, A. Nicolás, P. Peñil, E. Villar, F. Broekaert, M. Bourdelles, A. Cohen, M. T. Lazarescu, L. Lavagno, A. Terechko, M. Glassee & M. Prieto: "Improving the Design Flow for Parallel and Heterogeneous Architectures running Real-Time applications: The PHARAON FP7 project", Microprocessors and Microsystems,V.38, I.8, Part B, pp. 960–975, 2014.

[PRV11] 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.

[S3D18] “Single-Source System Modeling”, umlmarte.teisa.unican.es.

[SeGe14] B. Selic & S. Gerard: “Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems”, Morgan-Kaufman, 2014.

[TAH07] B. Tekinerdogan, M. Aksit & F. Henninger: “Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach”, Electronic Notes in Theoretical Computer Science, V.163, I.2, pp. 45-64, 2007.

[Tru06] F. Truyen: “The Fast Guide to Model Driven Architecture”, Cephas Consulting Corporation, 2006, https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf.

[TTOV97] LL. Teres, Y. Toroja, S. Olcoz and E. Villar: "VHDL: Lenguaje estándar de diseño electrónico", McGraw-Hill, 1997.

[Wel16] M. Weldon. “The future X Networks: A Bell Labs Perspective”, CRC Press, 2016.

[MMW11] W. Mueller, D. He, F. Mischkalla, A. Wegele, A. Larkham, P. Whiston, P. Peñil, E. Villar, N. Mitas, D. Kritharidis, F. Azcarate & M. Carballeda: "The SATURN Approach to SysML-Based HW/SW Codesign", N. Voros, A. Mukherjee, N. Sklavos, K. Masselos, M. Huebner (Eds.): "VLSI 2010 Annual Symposium Selected Papers", Lecture Notes in Electrical Engineering, V.57, pp. 151-164, Springer, 2011.

[MVH17] F. Mallet, E. Villar, F. Herrera: "MARTE for CPS and CPSoS", in S. Nakajima, J.P. Talpin, M. Toyoshima and H. Yu (Eds.): "Cyber-Physical System Design from an Architecture Analysis Viewpoint: Communications of NII Shonan Meetings", Springer, pp.81-108, 2017.