UML/MARTE Methodology for Heterogeneous System design

April, 2014

Microelectronics Engineering Group

TEISA Dpt. , University of Cantabria

Authors: P. Peñil

Index:

I. INTRODUCTION 10

Modelling Methodology 10

Definition of model views 12

Modelling process 13

II. MODEL VIEW SPECIFICATION 15

PIM VIEWS 17

1. DATA VIEW 17

1.1. Enumeration Data type 17

1.2. Primitive Data type 17

1.3. Derived Data type 18
1.3.1. Structure Data type 18
1.3.2. Array Data type 18

1.4. Specifying data types 19

1.5. Generalization of DataTypes 21
1.5.1. Data Type Generalization for Concurrency Exploration 21

2. FUNCTIONAL VIEW 23

2.1. Files 23

2.2. File specification 24

2.3. Refinement of files 24

2.4. Interfaces 25
2.4.1. Interface Services 25
2.4.2. Interface Inheritance 27

2.5. Libraries 29

2.6. Auxiliary Files 29

3. COMMUNICATION VIEW 30

3.1. Channel type specification 30
3.1.1. Storing Communication Mechanism 30
3.1.2. Communication semantics associated with a client application 30

3.2. **Synchronization Mechanisms** 32

3.3. **Shared Variable** 33

4. **APPLICATION VIEW** 34

4.1. **Application Components** 35
4.1.1. Application Component Attributes 35
4.1.2. Association of Files with Application Components 35
4.1.3. Association of File Folders with Components 36
4.1.4. The main application component 36
4.1.5. Ports 37
4.1.6. Connectors 37

4.2. **Application structure** 40
4.2.1. System ports: I/O communication 40
4.2.2. System Files 41
4.2.3. Libraries 41

4.3. **Files Folders** 42

4.4. **Modelling Variables** 42

4.5. **Modeling Variable Specification** 43
4.5.1. System Components 44
4.5.2. Language 45
4.5.3. Path 45
4.5.4. CFLAGS and LFLAGS 45
4.5.5. Compiler and Compiler path 45

4.6. **Application Components** 46

4.7. **Concatenation of paths** 46

5. **CONCURRENCY VIEW** 48

5.1. **Thread modeling** 48

5.2. **Thread structure** 49

5.3. **Application-Thread association** 49

5.4. **Initial function values** 50

6. **MEMORY SPACE VIEW** 51

6.1. **Process modelling** 51

6.2. **Process structure** 51

6.3. **Application Allocation structure** 52

PDM VIEWS 53

7. HW RESOURCES VIEW 53
7.1. HW Processors 54
7.2. Processor ISA 54
7.2.1. DSP processors 54
7.2.2. GPU processors 55
7.2.3. CPU co-processors 55
7.3. Processor Caches 55
7.4. HW Processor variables 55
7.5. Network 56
7.6. Network Interfaces 56
7.7. I/O Components 56
7.8. HW components' Functional Modes 56
8. SW PLATFORM VIEW 58
8.1. Drivers 59
8.2. Repository 60
8.3. Parameters 60
8.4. Device 60
PSM VIEWS 62
9. ARCHITECTURAL VIEW 62
9.1. Modelling of the HW/SW platform architecture 62
9.2. Allocation of SW instances to HW instances 63
9.3. Architectural Allocation 63
9.4. Allocation on DSP 64
9.5. Multiple HW resources allocation 65
9.6. Application Allocation to GPU 65
9.7. Thread allocation 66
9.8. Processor identifier 66
10. VERIFICATION VIEW 67
10.1. Environment components 67

10.2. Environment component Functionality 67

10.3. Environment component structure 68

10.4. Environment component structure: ports 68

10.5. Environment structure 69

10.6. Memory allocation 69

ANNEX 71

1. METHODOLOGY STEREOTYPES.. 71

2. METHODOLOGY ENUMERATIONS 75

Index of Tables:

Table 1 Data Specifier Values 20
Table 2 Data qualifier values 21
Table 3 Communication semantics to be implemented 32
Table 4 MARTE stereotypes used for refining the HW platform 53
Table 5 List of Stereotyes and attributes used in PHARAON methodology. 75

Index of Figures:

Figure 1 Design Flow 10

Figure 2 UML/MARTE view modelling activity 14

Figure 3 Model views 16

Figure 4 Enumeration data types 17

Figure 5 Primitive types 17

Figure 6 Structure Datatype 18

Figure 7 Array modelling 18

Figure 8 Array dimension specification by the Shape stereotype 19

Figure 9 Undef dimesion of an array 19

Figure 10 <<DataSpecification>> stereotype attributes 20

Figure 11 Data Generalizations 21

Figure 12 Data Type generalization for Concurrency exploration 22

Figure 13 Files 23

Figure 14 ApplicationFile stereotype attributes 24

Figure 15 Refinement of Files 25

Figure 16 Interfaces 25

Figure 17 Array size arguments 27

Figure 18 Interface generalization and operation1 of Interface1 27

Figure 19 Interface Inherence 28

Figure 20 Inheritance between interfaces 28

Figure 21 Libraries 29

Figure 22 Auxiliary *FilesFolder* packages 29

Figure 23 ChannelTypeSpecification stereotype attributes 30

Figure 24 Examples of Channel types 31

Figure 25 Notification resource 33

Figure 26 Shared variable 33

Figure 27 Application components. 35

Figure 28 Association Files-Application components 36

Figure 29 Associations of FileFolders with an Application Component 36

Figure 30 Main application component 37

Figure 31 Channel type attached to the *Channel* connector 38

Figure 32 Channel stereotype attribute 38

Figure 33 Assambly and delegation connectors 39
Figure 34 shared variable used by several application components 40
Figure 35 Application Structure 1 40
Figure 36 Application Structure 2 40
Figure 37 *System* component with files associated 41
Figure 38 *System* component with libraries associated 41
Figure 39 *System* component with *FileFolder* package 42
Figure 40 Specification of Variables 43
Figure 41 UML constraint for application component variables 43
Figure 42 Annotation in a UML constraint for variable specification 43
Figure 43 Example of multiple constaints in the same application component 44
Figure 44 Constrains of the “lmac” application component 44
Figure 45 Constraints with different constrained elements 44
Figure 46 $CFLAGs for native compilation 45
Figure 47 Compiler variable 46
Figure 48 Specification of the System’s base path 46
Figure 49 Application components with different types of model variables 47
Figure 50 Thread components 48
Figure 51 Thread instances which compose the thread structure 49
Figure 52 Generalization of the *System* component of the *Concurrency View* 49
Figure 53 Application-thread association 50
Figure 54 Memory partitions 51
Figure 55 Executables definition 51
Figure 56 Specialization of the *System* component of Memory Allocation View 52
Figure 57 Memory partition allocation 52
Figure 58 HW platform resources 54
Figure 59 HW Specification of a CortexA processor 55
Figure 60 HwProcessor compilers 56
Figure 61 HwProcessor mode specification 57
Figure 62 OS stereotype attributes 58
Figure 63 OS component 59
Figure 64 Driver for DSP management 60
Figure 65 “Parameter” driver property 60
Figure 66 “Device” driver property. 61

Figure 67 HW & SW platform architectures 63
Figure 68 Specialization of the *System* component of *Architectural View* 63
Figure 69 Memory partition allocation on HW/SW platform 64
Figure 70 Memory partition allocations to DSP 64
Figure 71 Application component allocation to a memory partition 65
Figure 72 Multi HwResources allocation 65
Figure 73 Application functions for GPU mapping 66
Figure 74 Function-GPU allocation 66
Figure 75 Thread-processor mapping. 66
Figure 76 Environment component 67
Figure 77 Environment application components 67
Figure 78 Environment Application components with associated Files 68
Figure 79 Application instances of an environment component 68
Figure 80 Environment Application components 69
Figure 81 Definition of the environment structure 69
Figure 82 Generalization of Environment structure with the *System* component of the *MemorySpaceView* 70
Figure 83 Allocation of environment component to the memory partitions 70

I. Introduction

As stated before, PHARAON specification is based on the UML/MARTE profile that provides features for real-time and embedded systems. In addition, PHARAON defines a profile that extends the UML/MARTE profile in order to offer some features for the specific description of the goals of the project.

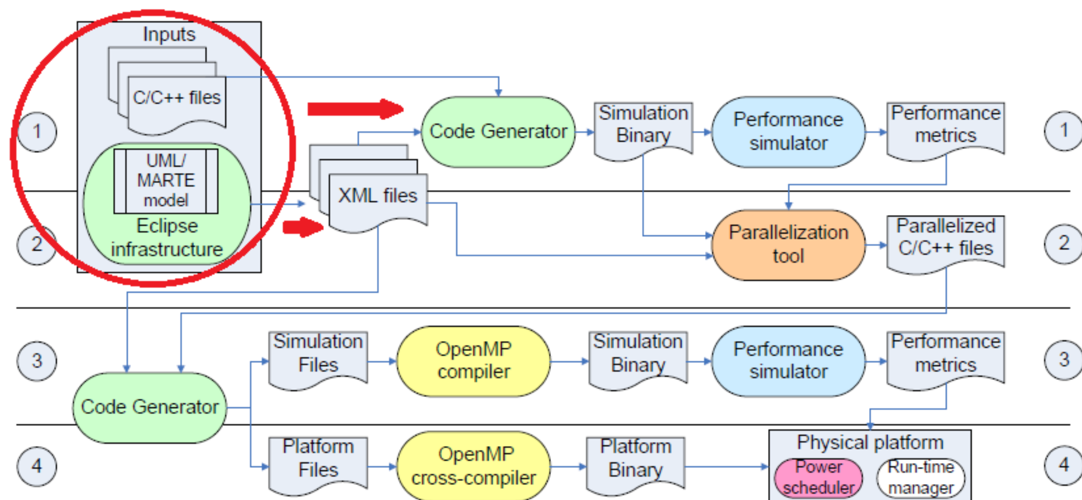

Figure 1 Design Flow

The system specification acts as the main input of the PHARAON project (Figure 1). As a result, the model created following the specification methodology must enable the designer to describe all the system characteristics required to obtain the optimal design in the physical platform. Thus, the model must combine the experience obtained in other projects and research activities, specially the COMPLEX project with the specific requirements specified for the tasks of synthesis, parallelization and run-time management.

As a result of that combination, the PHARAON partners have decided on the methodology and the modelling resources that will support the rest of the design activities in the project. The next subsections describe them.

Modelling Methodology

The complexity of embedded, parallel systems and platforms requires design methodologies that, based on separation of concerns, enable the design teams to work in an efficient way. Separation of concerns enables the specialization of the design process; separate but collaborating sets of designers can deal with different system concerns (application modelling, HW/SW platform design, etc), improving the development process. Therefore, well-defined system concerns in the same model enable designers to focus on their designing domain, guaranteeing system consistency

by using the same specification language, producing synergy among different design domains.

Support of this separation of concerns is covered in two steps. First, system models are divided into three sub-models, following the Y structure commonly applied in the latest design methodologies. In these flows, designs start with the definition of the two main starting points: HW platform and expected system functionality, and evolve to define how to support the functionality in the HW platform.

Following this structure, the system model is composed of different sub-models defined according to the features they must capture:

- The Platform Independent Model (PIM), which describes the functional and non-functional aspects of the system functions (e.g. application, functional code).
- The Platform Description Model (PDM), which describes the different HW and SW resources that form part of the system platform.
- The Platform Specific Model (PSM), which describes the system architecture and the allocation of platform resources.

Using these sub-models, the UML/MARTE system design activity takes charge of all modelling tasks required for initially defining the system under development, especially in the following aspects:

1. Data types
2. Modelling the code files.
3. Communication interfaces
4. Channel types
5. The system application, definition:
6. The functionality associated with each application
7. The concurrent structure of the application components
8. The communication media to interconnect the applications
9. Static threads associated with component operations
10. Memory partitions
11. The allocation of the applications into memory partitions
12. The system platform, both at HW and SW resource level.
13. The system architecture:
14. Instantiating HW/SW platform components,
15. Defining the allocation of the memory partition into the platform sources
16. The environment that interacts with the system

However, the integration of all these aspects into the three sub-models is too complex to be done. This is because UML models are based on graphical descriptions and so the number of elements that can be described in a model must be limited in order to maintain the benefits of the visual methodology. As a result, the three models are also sub-divided into parts, which are called views. Each of the previous modelling tasks are dealt with by using a model view.

The next sub-sections describe the views, what they are used for and the process defined to create them.

Definition of model views

There are different model views:

- **Data View**: defines the kind of data types used for the information exchange among the system functionalities. The view is mandatory.
- **Functional view**: this view includes the specification of the interfaces provided/required by the application components in order to be connected amongst themselves. Additionally, the view includes the specification of the files that contains the implementation (functional source code) of each application component. The view is Mandatoty.
- **Application View:** includes the definition of the application components and the application structure. Additionally, the view includes the association of the functional files defined in the *FunctionalView* with each application component. The view contains a "System" component that is used for specifying the application structure. It includes application components interconnected by using the interfaces defined in the *FunctionalView* and the communication mechanism defined in the *CommunicationView*. Mandatory.
- **Concurrency view**: this view includes all the threads of the system. Additionally, this view includes the association of the application components with these threads. The view contains a "System" component that is used for specifying the threads presented in the model and the mapping of the application components onto these threads. The view is mandatory.
- **Communication view**: captures the set of communication channels used for interconnecting the different application components. Additionally, the view includes the mechanisms used for synchronizing threads and processes. The view is optional if no communication media are considered.
- **Memory Space view**: defines the memory partitions that model the system processes as well as the allocation of application components onto these processes. The view is mandatory.
- **HW Resource view**: provides a description of the HW platform resources. The view is mandatory.

- **SW Platform view**: provides a description of the SW platform resources. The view is mandatory.
- **Architectural view**: defines the platform architecture and the mapping of system processes onto platform resources. Additionaly, this view includes the association of threads with processors. The view is mandatory.
- **Verification view**: defines the environment components that interact with the system. The view is not mandatory.

The **PIM** includes the views:

- Data View
- Functional view
- Application view
- Communication view
- Concurrency View
- Memory Space View

The **PDM** includes the views

- HW Resource view
- SW Platform view

The **PSM** includes the view:

- Architectural view

Modelling process

The following figure shows the proposed steps that are covered by the system specification methodology defined in this document. These steps guide the designer through the generation of the complete model required to perform the further synthesis and parallelization activities. The numbers in the boxes in the figure follow the proposed modelling step order.

The system specification methodology starts by defining the Model view where the designer models the data types required for the application and communication modelling. Then, four different views can be specified independently:

1. Data View: defines the data types used for specifying the arguments of operations and services of the application components
2. HW Resource view: describes the components of the HW platform. Each component specification must provide its identifier and its type, and the parameter values which define it.

3. SW Platform view: describes the components of the SW platform. Each component specification must provide its identifier and its type, and the parameter values which define it.
4. Verification View: defines the environment responsible for exciting the system. In a Test-Driven development methodology, the Verification view would be the first view to be defined. The following steps are recognized in this activity to achieve the previous objectives:
5. Identify the different environment subsystems interacting with the system and define how they interact with the system.
6. Model the functional elements that define the behaviour of the environment subsystems
7. Instantiate the system and the environment subsystems in order to define the environment-system structure.

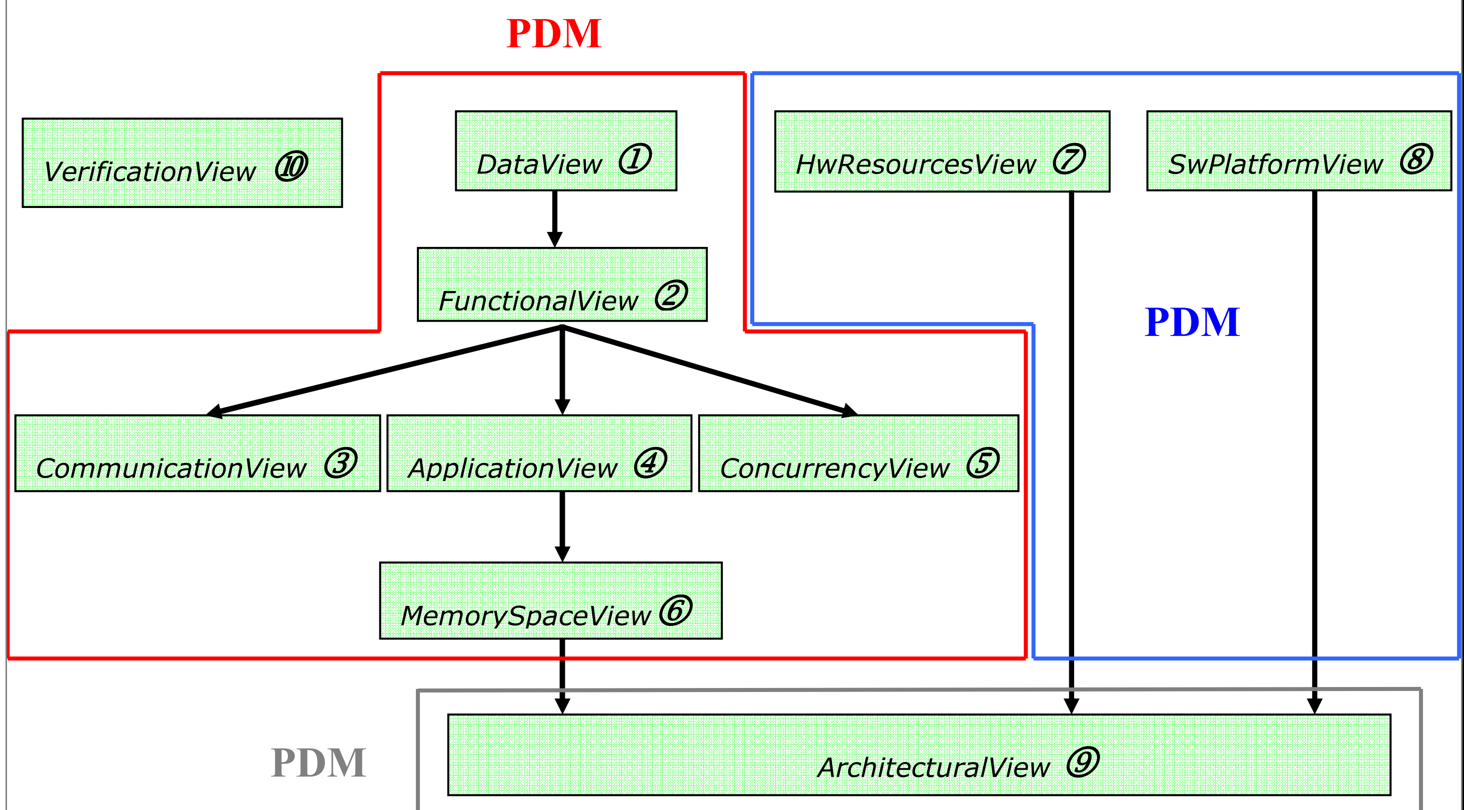

Figure 2 UML/MARTE view modelling activity

After defining the Data View, two model views can be defined in a cooperative way:

1. Functional View: defines the interfaces used by the applications to communicate, defining the operations that these interfaces have available and the arguments of these operations. The arguments are typed by the data types defined in the Data

Model. In addition, the functional elements that define application functionality are modelled.

2. Application view, which models the concurrency in the system in four ways:
3. Modelling the application components.
4. Associating the corresponding functional elements defined in the Functional view with each application component.
5. Define the application structure that consists of:
6. The specific structure of each application component defining the internal parts (application instances and the communicating elements defined in the Communication view that interconnects these applications)
7. The *top* of the application structure
8. Communication view: defines the type of channels used for communicating the application components.
9. Concurrency view: Modelling the static threads created when each application component is triggered and the operation executed by them.

In the next step, the components of the Concurency view are mapped to memory partitions. This task is dealt with in the Memory Allocation view where:

10. The memory partitions are defined.
11. The allocation of the *top* application instances defined in the Concurrency view are mapped to the memory partition instances.

The architectural view defines the HW platform architecture, by using instances of the HW components defined in the HW Platform view and the SW platform architecture by using instances of SW components defined in the SW Platform view. Then, the SW instances are allocated to HW instance resources. The second allocation process that takes place is the allocation of the instances of memory partitions defined in the Memory allocation view.

With the Architectural view, the system design is completed and the transformation process from UML/MARTE can be done.

II. Model View specification

As was mentioned before, 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 are:

<<DataView>>

<<FunctionalView>>

<<CommunicationView>>

<<ApplicationView>>

<<ConcurrencyView>>

<<MemorySpaceView>>

<<HwResourceView>>

<<SwPlatformView>>

<<ArchitecturalView>>

<<VerificationView>>

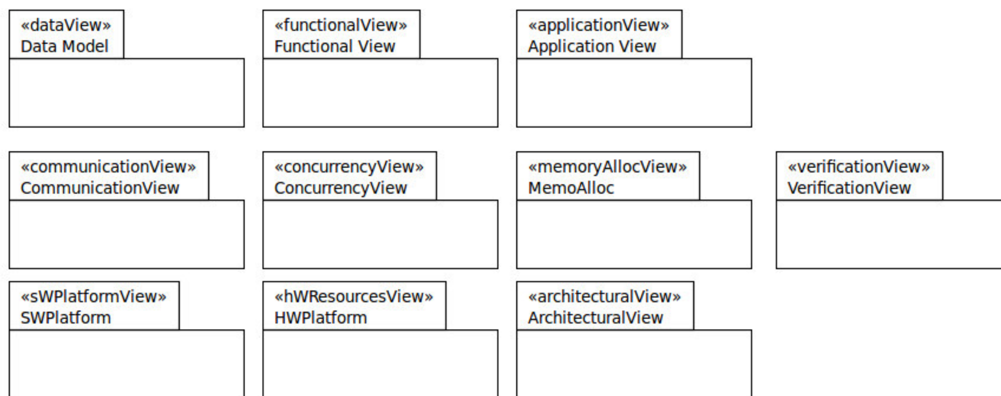

Figure 3 Model views

PIM Views

1. Data View

The data model view focuses on the modelling of the data types that will be involved in the interface services and application operations. These data types are included in UML class diagrams.

The data model view focuses on the modelling of the data types that will be involved in the interface operations. 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 long”, etc.) and UML Data Types that are used to define new 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 and UML Data Types.

1.1. Enumeration Data type

The enumerations are captured as UML Enumeration data types and the different values of the enumeration are modelled as Enumeration Literals (Figure 4).

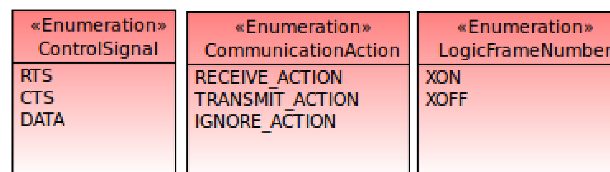

Figure 4 Enumeration data types

1.2. Primitive Data type

The UML PrimitiveTypes are used to define basic data types. As can be seen in Figure 5, all these data definitions are classic primitive data types in coding.

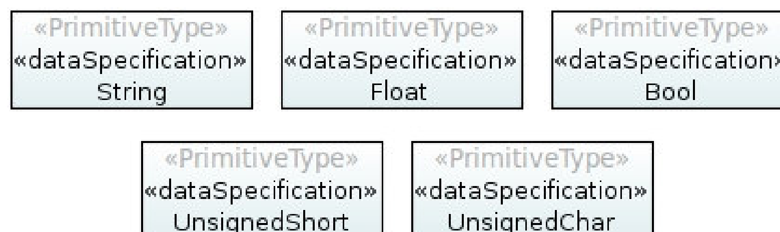

Figure 5 Primitive types

1.3. Derived Data type

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), structured data and arrays.

1.3.1. Structure Data type

Structured Data are modelled by using the MARTE stereotype <<TupleType>>. The *Datatype* has a set of properties typed by specific data type or primitive type that represent the fields of the structured data type.

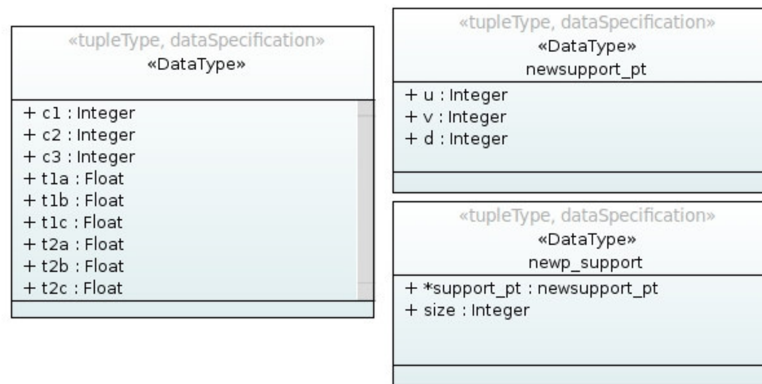

Figure 6 Structure Datatype

When a field of the structure data types is a pointer, an asterisk is annotated in the name ("newp_support" data type of Figure 6).

1.3.2. Array Data type

Arrays are modelled by using the MARTE stereotype <<CollectionType>>. The *collectionType* stereotype is applied to a *DataType* model element. A property is added to this DataType. This property should be typed by PrimitiveType or another DataType. Then, in the attribute *collectionAttrib* of the stereotype *CollectionType* that property should be attached (in Figure 7, property "array128i").

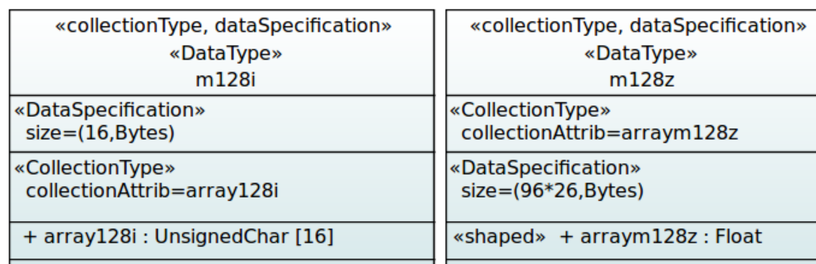

Figure 7 Array modelling

The dimension of the array is annotated in the multiplicity tag, if the array is unidimensional. 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 7 and Figure 8). 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 7).

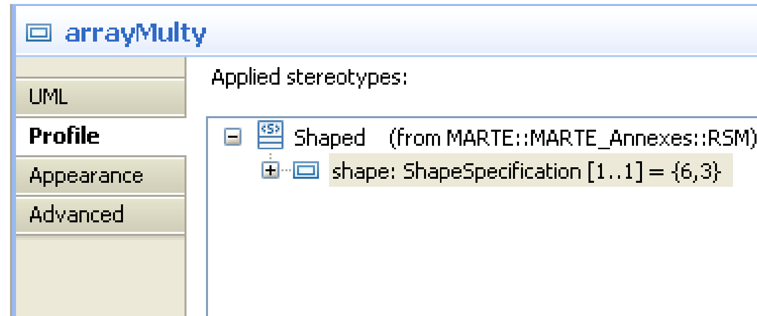

Figure 8 Array dimension specification by the Shape stereotype

In some cases, the designer can define the dimensions of an array with no specific value. Figure 9 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 multidimesion arrays (by appliying the stereotype *Shape*), the corresponding dimension should be specified by "*". Figure 9 shows these annotations.

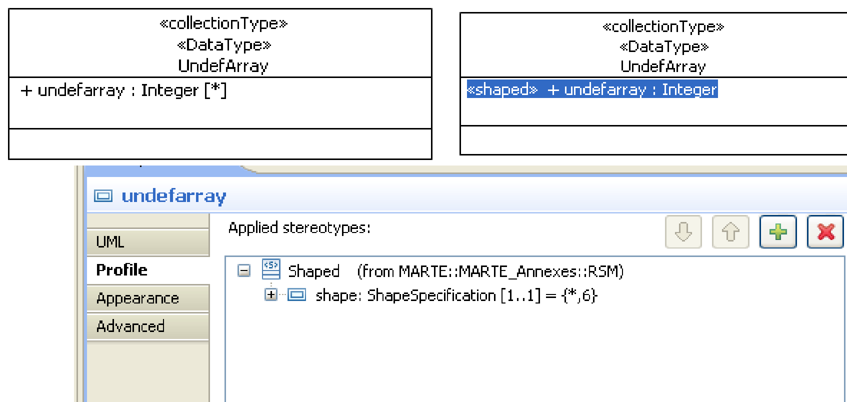

Figure 9 Undef dimesion of an array

1.4. Specifying data types

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

<<DataSpecification>>
size:NFP_Data [1]
pointer:Boolean [1]

dataSpecifier: DataSpecifier [1]
dataQualifier: DataQualifier [1]
complexDataType : String [0..1]

Figure 10 <<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* attribute: specifies whether the data is a pointer
- *dataSpecifier* attribute: denotes the C data specifier
- *dataQualifier* attribute: denotes the C data qualifier
- *complexDataType* attribute: 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**.

The list of values of the *DataSpecifier* attributes is:

<<Enumeration>> DataSpecifier		
None	signed int	long long int
Char	unsigned	signed long long
signed char	unsigned int	signed long long int
unsigned char	long	unsigned long long
short	long int	unsigned long long int
short int	signed long	float
signed short	signed long int	double
signed short int	unsigned long	long double
unsigned short	unsigned long int	void
unsigned short int	long long	
int		

Table 1 Data Specifier Values

The list of values of the *DataQualifier* attributes is:

<<Enumeration>> DataQualifier
None Const Volatile register

Table 2 Data qualifier values

1.5. Generalization of DataTypes

The modeling methodology enables the generalization of data types. If the *general* element of the UML generalization is a *Primitive Type* (in Figure 11, 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 general element of the UML generalization is a *Data Type* (in Figure 11, the data "Byte") the specific data is specified by the DataType (in Figure 11 the "QoS" is specified as "BYTE").

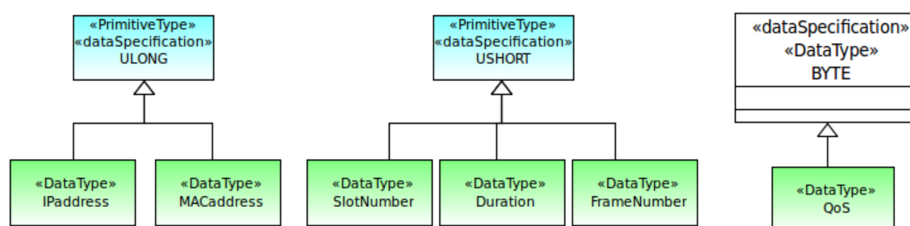

Figure 11 Data Generalizations

1.5.1. Data Type Generalization for Concurrency Exploration

In order to enable the exploration of the concurrency structure of the system, Data type generalization is required.

Some modelling constraints are applied to these data type generalizations:

- 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 specific element of the generalization (in Figure 12, the Data Type *DataType_Exploration*) should be specifed by the name of the

general element of the UML generalization (in Figure 12, the Data Type *DataType*).

- In the attribute *size* of the *DataSpecification* stereotype, the new and different value of the size (in Bytes) of data should be specifed.

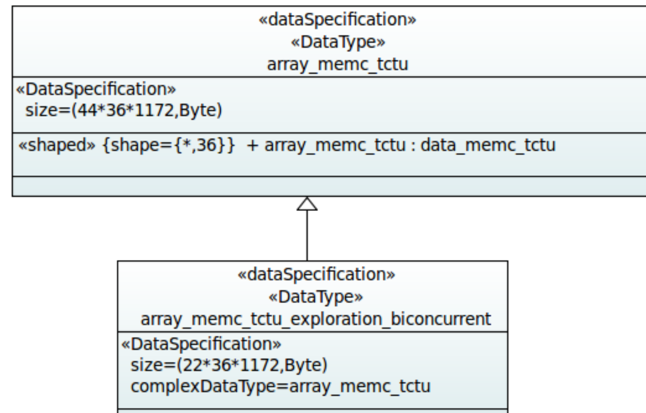

Figure 12 Data Type generalization for Concurrency exploration

2. Functional View

This view defines the functionality required/provided by the application components in order to exchange data for each particular functionality execution. This functionality is encapsulated in interfaces that are provided/required by the application components. At the modeling level, the same interface can be provided by different application components, although at implementation level these interfaces could be different.

Additionally, this view could include the set of files where the functionality performed by each application component is defined with C/C++ code.

The UML elements used in this view are:

1. UML Interfaces for modeling the application interfaces
2. UML Operations for modeling the interface services
3. UML Parameters for characterizing the interface services
4. UML Artifacts for modeling the files
5. UML comment for annotating deadlines

All these UML elements can be captured in Class diagrams. The next section will present the elements of the functional view of the proposed example.

2.1. Files

The files that store the implementation source-code of the applications are modeled by means of the UML element Artifact. 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, Figure 13).

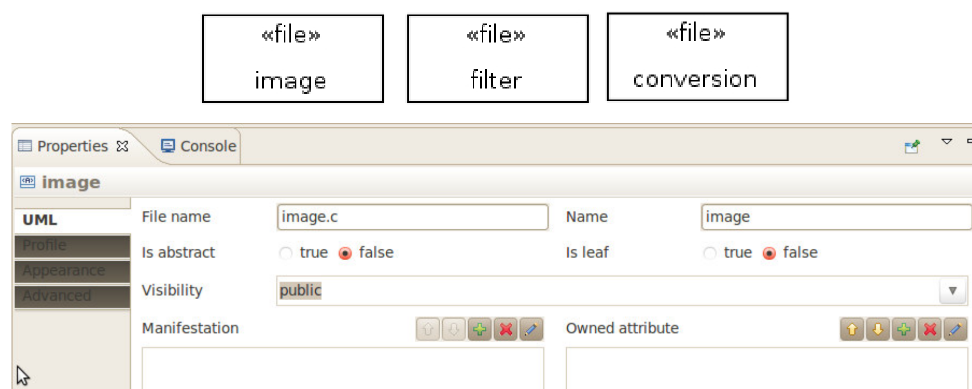

Figure 13 Files

2.2. 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:

6. **parallelized**: Boolean. The file is specified after the parallelization process.
7. **highLevel**: Boolean. The file coresponds to a high-level language not directly compilable (i.e Heptagon from which C can be optained).
8. **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.
9. **notModifiable**: Boolena. The file cannot be modified.
10. **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 14 ApplicationFile stereotype attributes

2.3. Refinement of files

Two different kinds of File artifacts can be defined: the artifacts only specified by the stereotype <<File>> and the artifacts specified by both stereotypes, <<File>> and <<ApplicationFile>>. In the first case, these files represent the functionality provided in the initial stage of the design flow. The combination of the stereotypes <<File>> and <<ApplicationFile>> means that the functionality of the corresponding artifacts has been refined for executing on a specific HW resource or that it has been modified by an external tool or by the user. In addition, the latter *files* can represent different file structures used for the different stages of the design process. In any case, the model should capture the relationship between the initial *files* and the refined *files*. This *file* refinement is captured by a UML Abstraction relationship between a *file* with a set of files. This UML abstraction is specified by the UML standard stereotype <<refine>>, as can be seen in Figure 15. Only one refined file is allowed for each design stage. There is one exception; when two files contain optimized code for a specific HW resource. For instance, two different implementations, one for a Neon execution and other one for a

DSP are shown in Figure 15. Depending on the HW resource where the application is mapped, the code generation annotates the correct file.

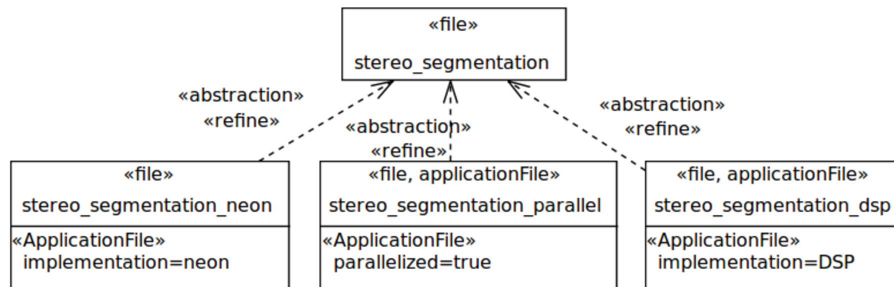

Figure 15 Refinement of Files

2.4. Interfaces

The interfaces capture the characteristics of the services provided/required by an application component in order to establish data exchange.

All the functions included in the same interfaces should be of the same type (sequential, guarded or concurrent). The same function can be included in different interfaces.

The application interfaces are modelled by means of UML interfaces. UML interfaces should be stereotyped by MARTE <<ClientServerSpecification>>. A *ClientServerSpecification* provides a way to define a specialized interface that allows its nature to be defined in terms of its provided and required operations.

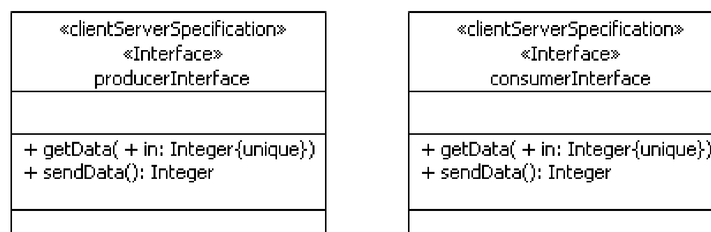

Figure 16 Interfaces

2.4.1. Interface Services

The interface services are modelled as UML operations. The functions can be:

- **re-entrant (sequential)**: no concurrency management mechanism is associated with the functions and, therefore, concurrency conflicts may occur. It is modelled by specifying the UML operation as *sequential*.
- **protected (guarded):** multiple invocations of the function may occur simultaneously at one instant but only one is allowed to commence. The others are blocked until the performance of the currently executing function invocation is complete. It is modelled by specifying the UML operation as *guarded*.

- **not re-entrant** and **not protected (concurrent)**: multiple invocations of a function may occur simultaneously at one instance and all of them may proceed concurrently. It is modelled by specifying the UML operation as *concurrent*.

Service Arguments

The 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 appliying the stereotype <<Pointer>>, the parameter is defined as a pointer.

Reference

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

Qualifier

The function arguments can be specifed by a qualifier by appliying the stereotype <<ParameterQualifier>>. Values associated with the *ParameterQualifier* stereotype are “const”, “volatile” and “register”.

Parameter of a array size

In the functions that a parameter is typed by an array data type, the function declaration can include parameters which are associated with the size of the arrays (“parameters-array_size”). In order to connect the “size” parameter with the corresponding “array” parameter, in the attribute *Default Value* of the “parameters-array_size” should include?? (Figure 17):

- name_parameter_array.length_x()
- name_parameter_array.length_y();
- name_parameter_array.length_z();

A parameter of the same “size” can be used for specifying the size of different arrays. In this case, each array reference should be separated by a semicolon (Figure 17).

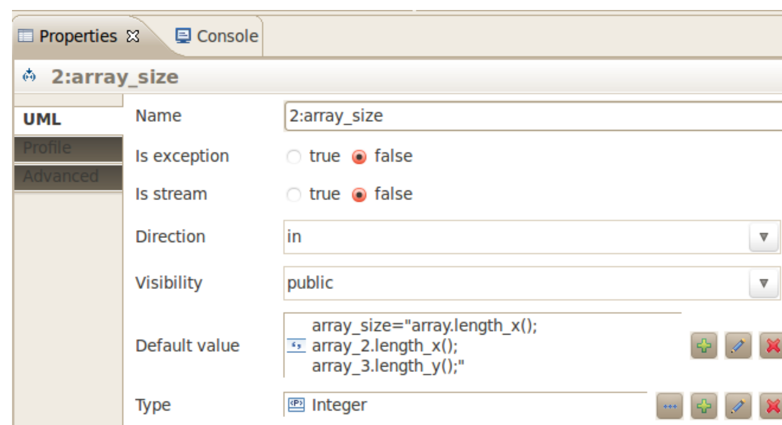

Figure 17 Array size arguments

2.4.2. *Interface Inheritance*

The methodology enables interface inheritance. This inheritance allows the redefinition of operations of the interfaces partitioning the sizes of the data streams sent and received. These streams are described in the model as parameters of these operations. The interface inheritance enables the definition of different concurrent structures in order to explore different design alternatives.

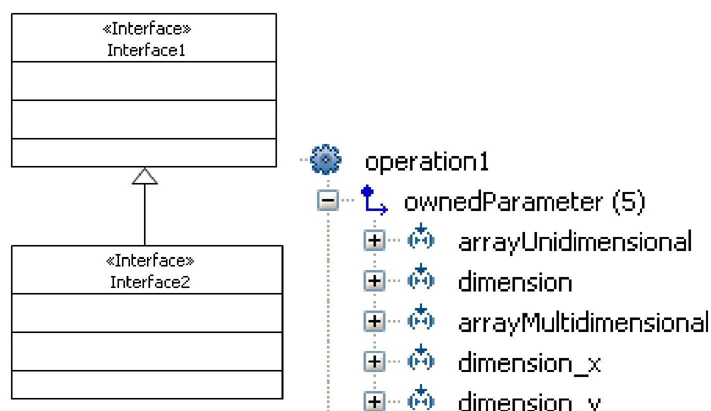

Figure 18 Interface generalization and operation1 of Interface1

All the functions of these interfaces are the same and have the same parameters (with the same name and order). For the data partitioning, only the parameter to be used for tha data splitting is necessary to be specified. The only difference is that one parameter is specified by different data types. This new data type is a generalization of the previous data type (see section on Data Types for exploration of concurrency structure).

Several parameters of a same function can be used for data splitting (Figure 19).

Several parameters of functions of a same interface can be used for data splitting.

The functions of an interface that are not used for data splitting should not have any parameters (Figure 19).

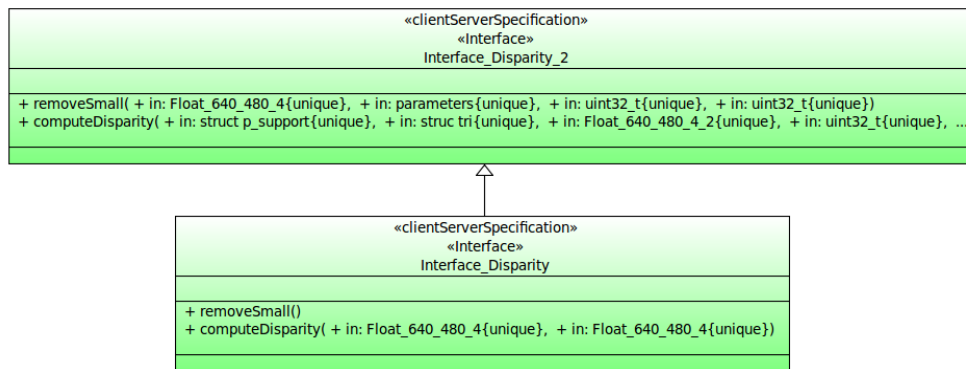

Figure 19 Interface Inherence

Additionally, interface inherence is used to join different concurrent flows. This is explained through an example. The interfaces model the functions provided by the application components for enabling the applications interconnections. In this case, a different interface is used. In communication of the components "matchingRight", "matchingLeft" and "stereoMatching" three different interfaces (Figure 20) are used. All these interfaces have the same function associated, "stereo_matching". However, the declaration of this function in these interfaces is totally different. In the interface "Interface_StereoMatching" the function "stereo_matching" is completely specified, where all the properties of the function parameters are completely characterized (data type, size, pointer, etc.). On the other hand, the functions of the other interfaces ("Interface_StereoMatching_Right" and "Interface_StereoMatching_Left") only specify the parameters used for joinning. Specifically, in the "stereo_matching" two parameters are used to join both concurrent flows: "img_left" and "img_right". In order to be executed, the component "stereo_matching" has to be available for both images pre-processors. However, the two images come from two different, independent, concurrent flows. In order to specify that a parameter represents an element to be joined, the corresponding join parameters have to be specified in the generalized interfaces; only these parameters have to be specified in generalized interfaces (in the example, "img_left" and "img_right"). Then, these parameters are not typed by any data type which it is understood by the code generator that the parameter is for joining concurrent flows. In the case of Figure 20, the function "stereo_matching" of the interface "Interface_StereoMatching_Right" only includes the parameter "img_left" which denotes that this parameter should be provided by other components and, therefore, the "stereo_matching" has to wait for it. For the interface "Interface_StereoMatching_Left", the parameter specified and not typed is "img_right".

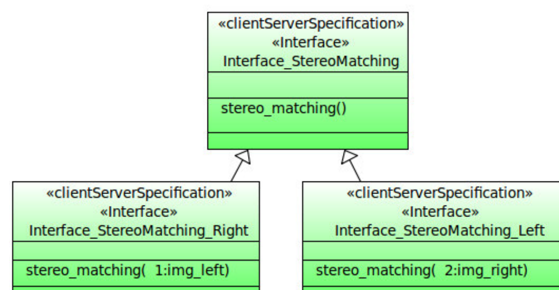

Figure 20 Inheritance between interfaces

2.5. Libraries

In order to enable the compilation of the application, a set of specific libraries can be necessary. Therefore, in order to enable the generation of the makefiles, these libraries should be modeled. These libraries are modeled as UML Artifacts specified by the UML standard stereotype <<library>> (Figure 21).

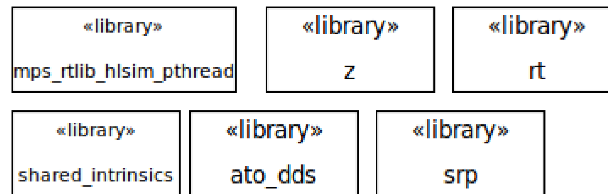

Figure 21 Libraries

The Library artifacts can only be associated with the *System* component included in the *ApplicationView*.

2.6. Auxiliary Files

As was described previously, each application component has the files that implement each specific application functionality associated. However, this 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:

1. **parallelized**: the file folder contains files produced after a parallelization process.
2. **highLevel**: the file folder contains files that specify high-level functionality.
3. **implementation**: the file folder conatins files which are optimized to be executed in a specific HW resource: **DSP. NEON, GPU).** .
4. **notModifiable** : the file folder contains files which cannot be modified for any reason.
5. **environment**: the file folder contains a test bench.

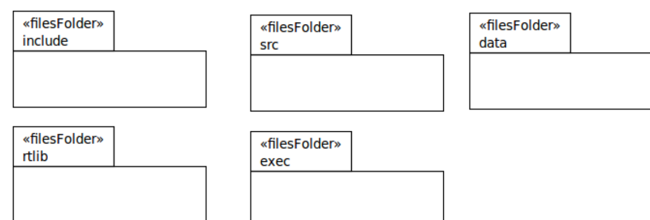

Figure 22 Auxiliary *FilesFolder* packages

The *FilesFolder* package can be associated with application components (*RtUnits*) and to the *System* components included in the *ApplicationView*.

3. Communication View

The Communication view defines the communication mechanisms that enable the application components' communication.

The UML element used in this view is the UML Component used to model the communication components. Class diagrams are used for defining these communication components.

3.1. Channel type specification

The generic communication mechanism is modelled by the MARTE stereotype <<CommunicationMedia>> that represents the means to transport information from one location to another. Then, new characteristics can be added to the communication media in order to define a different communicatuion semantics

3.1.1. Storing Communication Mechanism

A *CommunicationMedia* can be specified with additional characteristics in order to define different communication semantics. The *CommunicationMedia* could have the capacity to store function call requests. To model this characteristic, the MARTE stereotype <<StorageResource>> can be applied to the *CommunicatinMedia*. The attribute *resMult* of the *StorageResource* denotes the number of function call requests that can be stored.

3.1.2. Communication semantics associated with a client application

Some additional characteristic can be added to the communication media in order to model the communication semantics associated with the application component that uses the communication media to access a service provided by another application component.

The stereotype <<ChannelTypeSpecification>> adds additional characteristics to the communication media in order to model different communication semantics.

<<ChannelTypeSpecification>>
blockingFunctionDispatching:Boolean [1]
blockingFunctionReturn:Boolean [1]
priority : integer [0..1]
timeOut:NFP_Duration [0..1]
ordering : Boolean [1]

Figure 23 ChannelTypeSpecification stereotype attributes

The attribute *blockingFunctionDispatching* defines the behaviour of the client application when it requires a service from a server application: the client application is blocked until the server application attends to the service request.

The attribute *blockingFunctionReturn* defines whether the client application is blocked waiting for the response from the service called.

The attribute *priority* defines the priority associated with client-application client in order to attend service requests coming from the channel.

The attribute *time out* defines the maximum time for waiting for a function's call response.

The attribute *ordering* defines whether the concurrent calls transmitted through the channel have to be synchronized in the function return and to be dealt with as an ordered set.

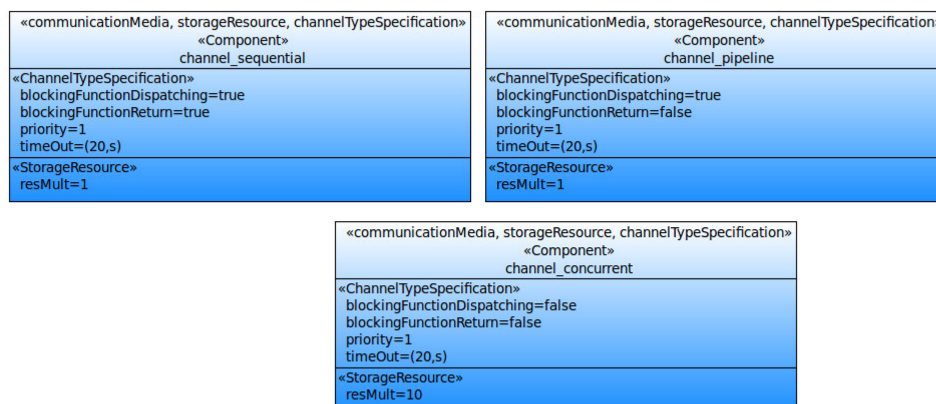

Figure 24 Examples of Channel types

The following table describes the possible semantics that can exist depending on the values of the attributes *blockingFunctionDispatching, blockingFunctionReturn* of the stereotype <<ChannelTypeSpecification>>, the *resMult* attribute of the MARTE stereotype <<StorageResource>> and the attribute *srPoolSize* of the MARTE stereotype <<RtUnit>> (explained in the next section). Additionally, the table specifies the behaviour of the function call communication during execution time. The table denotes:

1. Capacity available in execution time.
2. Value of the attribute *blockingFunctionDispatching*.
3. Value of the attribute *blockingFunctionReturn*.
4. Service threads: the application component has threads available in order to attend to service requests.
5. Store, the function call request should be stored or not in the channel
6. Block call, the client should be blocked bnefore dispatching its function call request
7. Block return, the client should be blocked waiting for finalization of the function called.

8. Exec, the function called can be executed or it should be delayed until resources are available.

	Capacity	Blocking Function Dispatching attribute	Blocking Function Return attribute	Service threads	Store	Block Call in the client for no channel capacity available	Block for data return	Exec when thread Not available
1	available	true	true	available	No	No	Yes	Yes
2	available	true	true	Not available	Yes	No	Yes	Delay
3	available	false	true	available	No	No	Yes	Yes
4	available	false	true	Not available	Yes	No	Yes	Delay
5	available	false	false	available	No	No	No	Yes
6	available	false	false	Not available	Yes	No	No	Delay
7	available	true	false	available	No	No	No	Yes
8	available	true	false	Not available	Yes	No	No	Yes
9	Not available	true	true	available	No	No	Yes	Yes
10	Not available	true	true	Not available	No	Yes	Yes	Delay
11	Not available	false	true	available	No	No	No	No
12	Not available	false	true	Not available	No	No	No	No
13	Not available	false	false	available	No	No	No	No
14	Not available	false	false	Not available	No	No	No	No
15	Not available	true	false	available	No	No	No	Yes
16	Not available	true	false	Not available	No	Yes	No	Delay

Table 3 Communication semantics to be implemented

3.2. Synchronization Mechanisms

To model the synchronization mechanisms among application components, the MARTE stereotype <<NotificationResource>> is used. *NotificationResource* supports control flow by notifying awaiting concurrent resources about the occurrence of conditions.

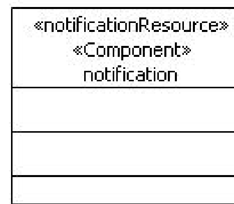

Figure 25 Notification resource

3.3. Shared Variable

The two previous communication mechanisms can be used to connect application components that are allocated in the same memory partition or in different memory partitions. An additional communication mechanism can be used in order to enable the communication among application components. This communication mechanism is the shared variable. The shared variable is modelled by the MARTE stereotype <<SharedDataComResource>>. *SharedDataComResource* defines a specific resource used to share the same area of memory among concurrent resources to exchange information by reading and writing in this area of memory.

The shared variable can be protected or not. To model a protected variable the stereotype attribute *isProtected* should be used. For specifying the type of the shared variable, a UML property should be included in the UML Component *SharedDataComResource*. The type should be included in the *DataView*. Then, in the stereotype attribute *identifierElements* this property should be attached.

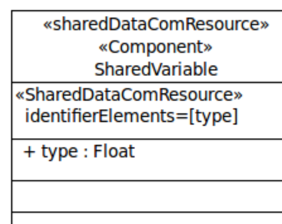

Figure 26 Shared variable

4. Application View

This view defines the different software components that are used to build the system application. The complete system application is composed of instances of these application components interconnected through ports by channels using interfaces defining the functionality required/provided by the application components. Interfaces represent the set of functions that are required/provided by an element from/to its specific environment. Channels are communication media for connecting application instances that can have associated communication properties for characterizing data transmitted.

Additionally, this view includes:

- association of *Files* included in the *FunctionalView* with the application components
- association of the *FilesFolders* with the application components and the *System*
- specification of the application *System* structure, composed of instances of interconnected application components
- association of the libraries with the *System*

The UML elements used in this view are:

1. UML Component for modeling the application components and for defining the element where the complete application structure is captured
2. UML Port are the interaction points between the component and its environment
3. UML Connectors for connecting application component instances. These connectors are channels with specific communication semantics
4. UML Operations for defining internal functions of the application components
5. UML Parameters for characterizing the internal functions of the application components
6. UML Abstraction for associating *Files* defined in the *FunctionalView* with the application components
7. UML constraint for defining paths, flags, compilers, etc.
8. UML links for associating constraints with model elements

Class diagrams are used for defining the application components and associating *Files, FilesFolder* and constraints with application components.

Class diagrams are used for associating *Files, FilesFolders, Libraries* and constraints with *System* components.

Composite structure diagram is used for defining the structure of the application system.

4.1. Application Components

The application components are modelled by the MARTE stereotype <<RtUnit>> (Figure 27). *RtUnit* component has its own execution threads, its associated C files, providing/requiring services to/from other application components by means of provided and required interfaces. These provided/required interfaces and C files were defined in the *FunctionalView*. Additionally, a *RtUnit* component can have an associated set of theads in order to execute some specific functions concurrently.

4.1.1. Application Component Attributes

The following attributes of the <<RtUnit>> stereotype are considered (Figure 27):

- The attribute *isDynamic* and *srPoolsize*. In this methodology, we defined that the value of attribute *isDynamic* should be true to specify that the application component creates threads dynamically in order to attend to the requests for services provided by the *RtUnit*.
- The attribute *srPoolSize* should be defined by a specific value in order to denote that the *RtUnit* has a finite set of threads to attend to the request for the services provided by the *RtUnit*.
- The attribute *srPoolPolicy* should be *infiniteWait* to denote that the *RtUnit* waits infinitely till a thread finishes attending to a service request if the RtUnit does not have more threads available.
- The *isMain* attribute can be considered to denote the main application which is used for generating a specifc file for the synthezesing code.

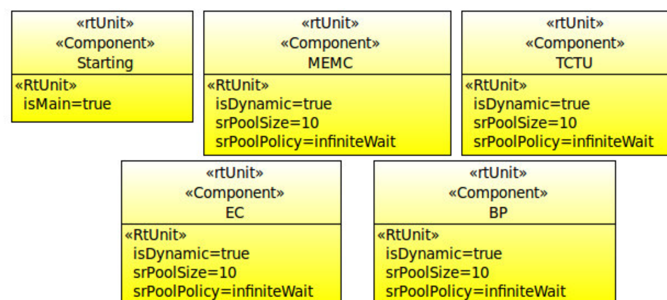

Figure 27 Application components.

4.1.2. Association of Files with Application Components

The specification of the set of files associated with an application component is defined:

- By using an UML Class diagram
- By using the *File* UML artifacts (code files) defined in the *FunctionalView*.

The code files are associated with a *RtUnit* component by means of an UML abstraction specified by the MARTE profile <<Allocated>> (Figure 28).

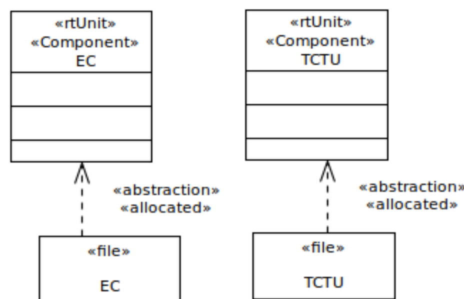

Figure 28 Association Files-Application components

4.1.3. Association of File Folders with Components

The application componets can have associated FileFolders. These *FilesFolder*s are associated with the application components such as *Files*: by using a UML abstraction specified by the stereotype <<allocated>>.

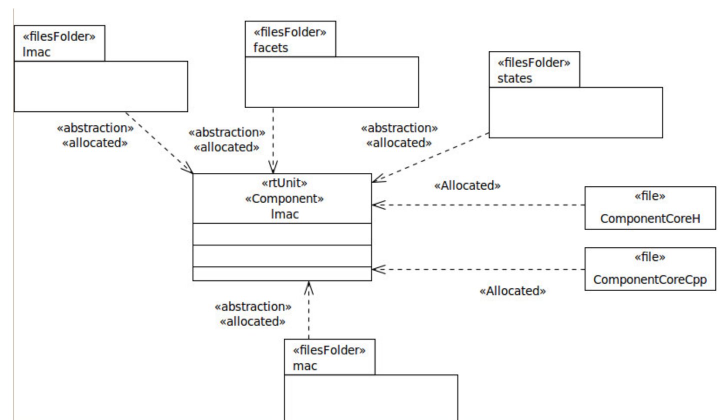

Figure 29 Associations of FileFolders with an Application Component

4.1.4. The main application component

The main application component is identified by the *RtUnit* attribute *ismain*, specified as “true”. Thus, this *RtUnit* component should have an associated UML operation. This UML operation should be given the same name as the main procedure of the functionality. This UML operation should be associated to the *RtUnit* component through the *RtUnit* attribute *main*.

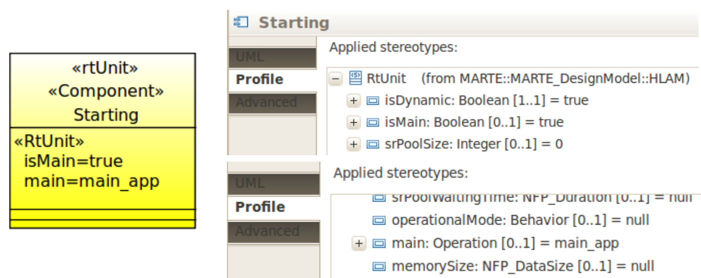

Figure 30 Main application component

4.1.5. Ports

Communication among application components is established through UML ports. The ports denote the services encapsulated in the interfaces that the application component required or provided. These ports must be modeled in different ways depending on the type of communication:

When communication is by means of function calls of interfaces, the UML ports 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. In the attributes *provInterfaces* and *reqInterface,* the interface required or provided by the port is defined. Only one interface can be attached to the *ClientServerPort*. The *ClientServerPort* can be either *provided* or *required*.

In other communication mechanisms, the UML (shared variable and synchronization mechanism) ports should not be specified by any stereotype.

4.1.6. Connectors

The ports are connected by using UML connectors. The conectors can represent simple connections or communication channels.

The former defines the connection between an application element and a shared variable. Additionally, in a communication based on interfaces, a simple connector denotes a pure RCP (Remote Call Protocol) in the client-server communication paradigm.

<u>Channels</u>

The connectors among the application elements can respresent specific communication channels with a well-defined semantics. In this case, the UML connectors should be specified to define the semantics of communication established among the application components. The stereotype <<Channel>> enables the specification of a UML connector by a communication mechanism defined in the *CommunicationView*. The attribute of the *Channel* stereotype is *channelTypeSpecification,* which defines the communication mechanism specified in the *CommunicationView*. In this attribute, a channel type captured in the *CommunicationView* is attached (Figure 31).

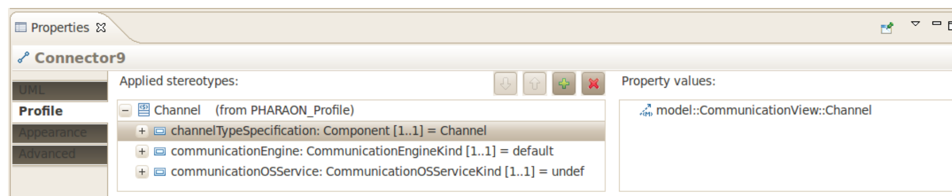

Figure 31 Channel type attached to the *Channel* connector

Each UML connector specified by the *Channel* stereotype identifies a different communication channel to be implemented.

In order to capture this implementation information, the <<Channel>> stereotype is associated with additional attributes. The attribute *communicationEngine* is an enumeration with a set of communication libraries independent of the platform. The possible values are *MCAPI*, *OpenMP*, *OPenStream*, *TCP/IP* and *default are;*

- *MCAPI* is a standard communication API for distributed embedded systems.
- *OpenMP* is a library for multi-processor programming of shared memories.
- *OpenStream* is a data-flow extension of OpenMP to express dynamic dependent tasks.
- *TCP/IP* protocol of data transmission.
- *undef* means the previous communication mechanism is not used.

A second attribute of the *Channel* is *communicationOSService*. This attribute is an enumeration that denotes different communication mechanisms provided by an OS. The possible values are *FIFO* channels, *sockets*, *message queues*, *shared memories*, *files*.

When the values of the attributes *communicationEngine* and *communicationOSService* are *undef* and *default* respectively, it means the communication mechanism implemented for a channel derives from the OS where the interconnected application components are mapped.

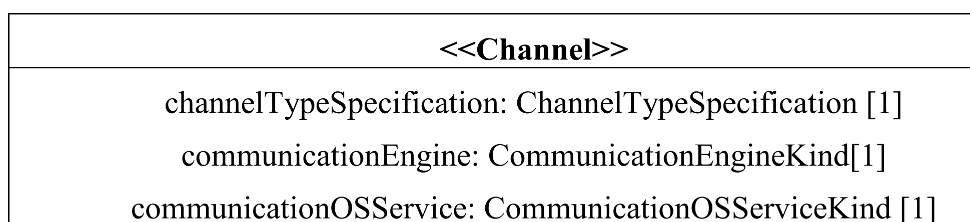

Figure 32 Channel stereotype attribute

Only UML assembly connectors (in Figure 33 the UML connector established between the elements “imageAcquisition” and “imagePreProcessing”) should be stereotyped by the *Channel*. The *UML delegation* connectors (in Figure 33 the UML connectors that interconnect the “imageAcquisition” ports “port_Condev”, “port_disDev” and “portCapImage”) with ports of the *System* which establishes communication with the environment.

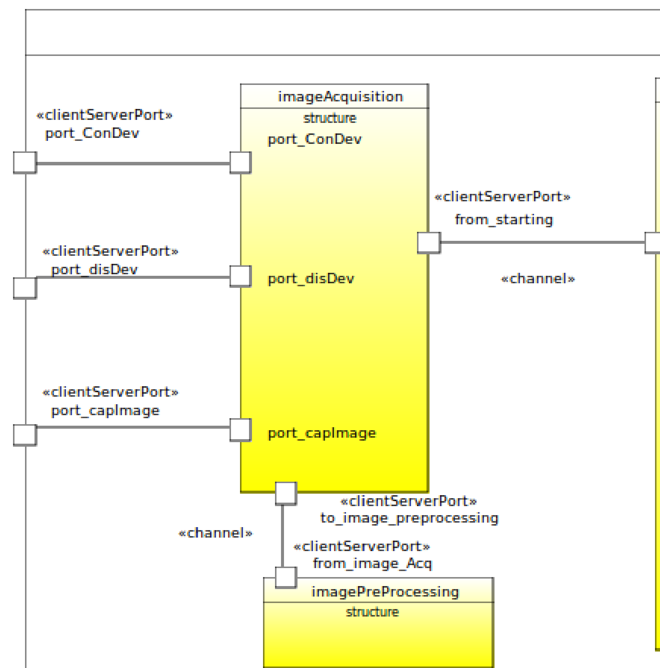

Figure 33 Assambly and delegation connectors

Communication Mechanism and Interfaces

The previous communication mechanisms enable the information exchange among applications through function calls provided by interfaces. The same interface can be provided by different application components or can be provided through different ports by the same application interfaces. The *Channel* connectors that are associated with the same interface represent the same channel in the implementation stage. Therefore, these *Channel* connectors should be typed by the same communication media defined in the *Communication View,* thus ensuring the model coherence: the communication media should have the same interface associated with the application ports.

Connection through shared variables

A shared variable can be used for communicating two or more application components. For connecting application components with the same shared resource, an instance of a *SharedDataComResource* has to be included in the composite structure diagram of the *System* component of the *ApplicationView*. Then, the application components are connected to this *SharedDataComResource* instance by using UML connectors (Figure 34).

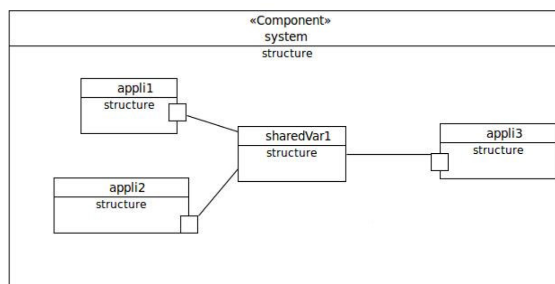

Figure 34 shared variable used by several application components

4.2. Application structure

In order to define the top application structure, a UML component is used. This component is specified by the stereotype <<System>>. This *System* component constains instances of the *RtUnit* application components interconnected by using connectors that can represent communication channels or shared variable accesses.

The application structure is captured in a UML Composite Structure diagram associated with the System component.

Only one *System* component should be defined within the *ApplicationView* package.

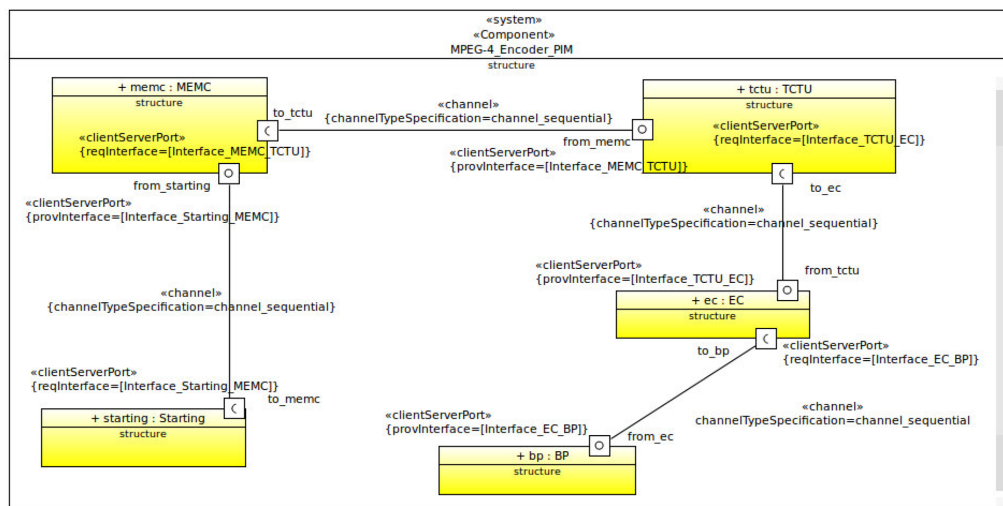

Figure 35 Application Structure 1

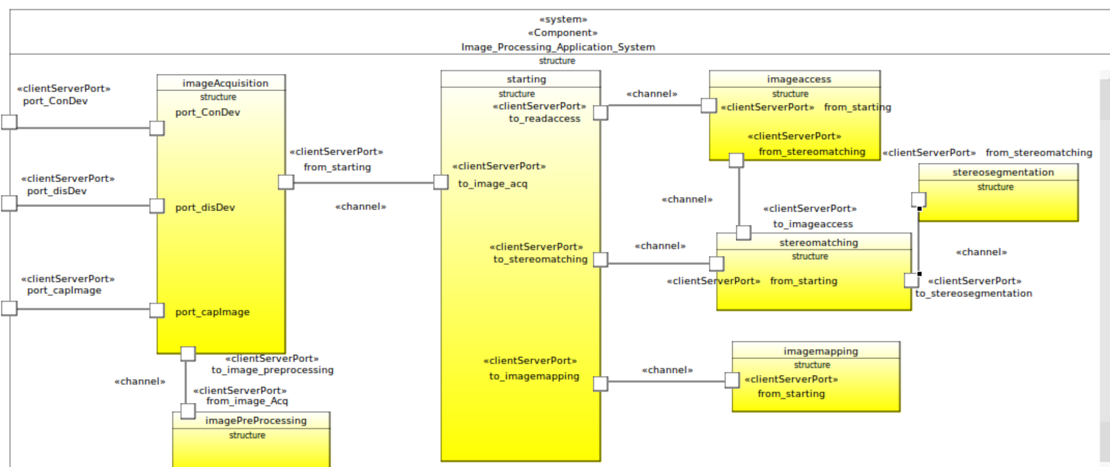

Figure 36 Application Structure 2

4.2.1. System ports: I/O communication

The *System* communicates with the external environment. This environment communication is established through ports. These UML ports should be specified by

the MARTE stereotype <<ClientServerPort>> (Figure 33), specifying the correct values of the attribute *kind, provInterface* and *reqInterface* depending on the communication is dealt with using function calls. If not, the ports should not be stereotyped.

These *System* ports connect with an application instance. This connection is port-to-port and the name of the application component port has to be the same as the *System* port (Figure 33). The application component port are not stereotyped. The connection between the System port and the application port is never stereotyped (Figure 33).

4.2.2. System Files

The *System* component may have associated *files*. These files are defined in the *FunctionalView* and identified by the UML standard stereotype <<File>> and by the PHARAON stereotype <<SystemFile>>. These files are associated with the *System* component through a UML abstraction specified by the MARTE stereotype <<allocated>>, as is shown in Figure 37.

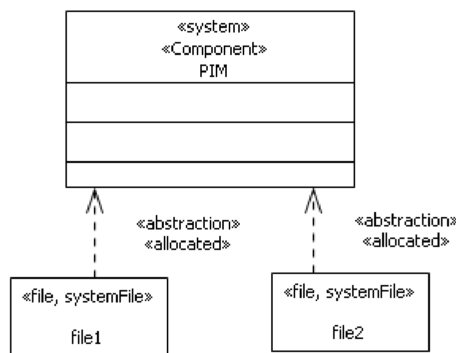

Figure 37 *System* component with files associated

4.2.3. Libraries

In order to enable the compilation of the application, a set of specific libraries can be required in order to enable the makefiles' generation

The *Libraries* defined in the *FunctionalView* are associated with the *System* component by means of UML Use relations, as Figure 38 shows.

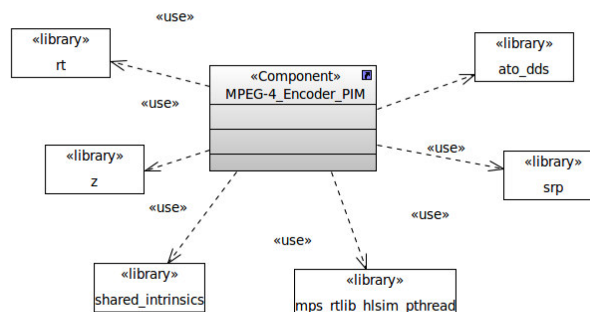

Figure 38 *System* component with libraries associated

4.3. Files Folders

The *FilesFolders* packages defined in the *FunctionalView* are associated with the *System* component by a UML abstraction association specified by the MARTE stereotype <<Allocated>>. The designer is free to include the corresponding UML artifact files in these packages in order to model the real auxiliary files explicitly; this is not mandatory.

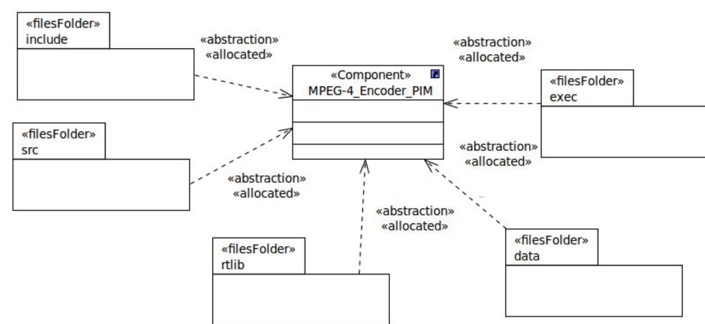

Figure 39 *System* component with *FileFolder* package

4.4. Modelling Variables

The model has modelling variables. More specifically, in the modelling of the application, these modelling variables are used to define characteristics required for completely characterizing the application components of the system in relation to the makefiles' generation and code generation. The modelling variables are:

1. *language*: specifies the language in which the specific application functionality is implemented. Not mandatory (by default, it is "C").
2. *path*: specifies the path where the functional files are allocated in the host. Mandatory for the *System* component.
3. *path_system*: specifies a path of a *File* or *FilesFolder* of a application component that has as first part of the absolute path, the path associate to the *System* component
4. *creation*: specificies the mechanism used to create a specific application component instance. Mandatory only when the language is "C++".
5. *cc_compiler*: specifies the C compiler.
6. *cxx_compiler:* specifies the C++ compiler.
7. *path_compiler:* specifies the path where the compiler (C or C++) is allocated.
8. *CFLAG*: defines the compilation flags
9. *LFLAG*: defines the linking flags.

4.5. Modeling Variable Specification

The variables are annotated as *$nameVariable="valueVariable";* as Figure 40 shows.

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

Figure 40 Specification of Variables

The model variables are annotated with UML Constraints owned by the component (*RtUnit, System, etc.*) denoted in the ownedRule of the component (Figure 41) and in the "Context" attribute of the constraint (Figure 41).

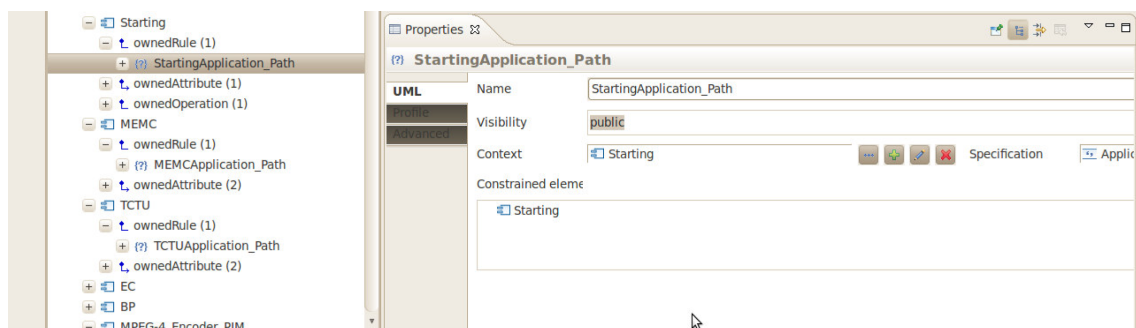

Figure 41 UML constraint for application component variables

The "Specification" attribute of constraint contains the declaration of the variables. The variable annotation is captured in a LiteralString (Figure 42).

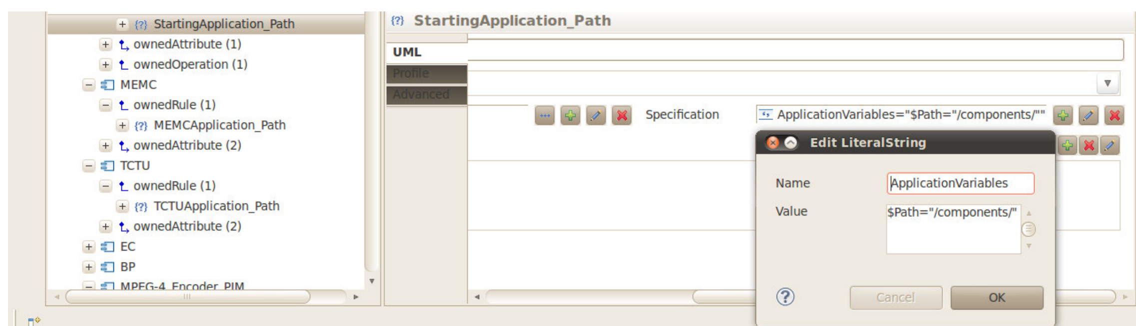

Figure 42 Annotation in a UML constraint for variable specification

Then, the constraint is associated with an element model that is included in the ConstrainedElement attribute of the UML constraint (Figure 41). 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. In Figure 43, there are 4 constraints ("MAC_LMAC_states_facets", "MAC_LMAC_varibles", "MAC_InterfacesFolder_LMAC_common" and "MAC_Folder_LMAC").

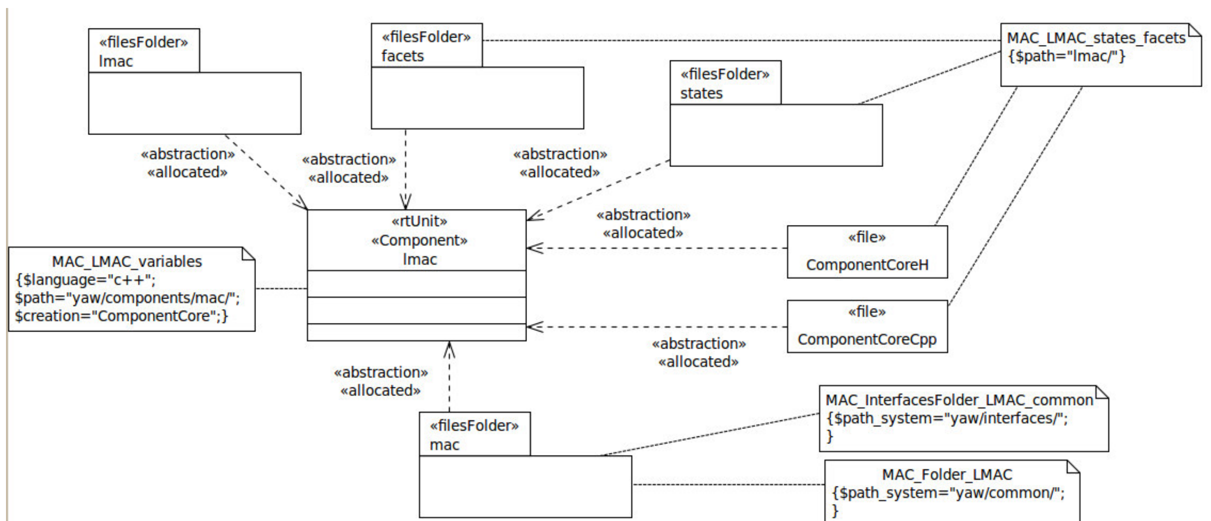

Figure 43 Example of multiple constaints in the same application component

All these UML constraints are owned by the application component "lmac" (Figure 44).

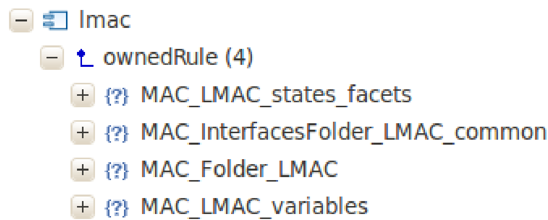

Figure 44 Constrains of the "lmac" application component

However, not all of these constraints are applied to the same model element, denoted by the attribute "ConstrainedElement" of the constraints (Figure 45).

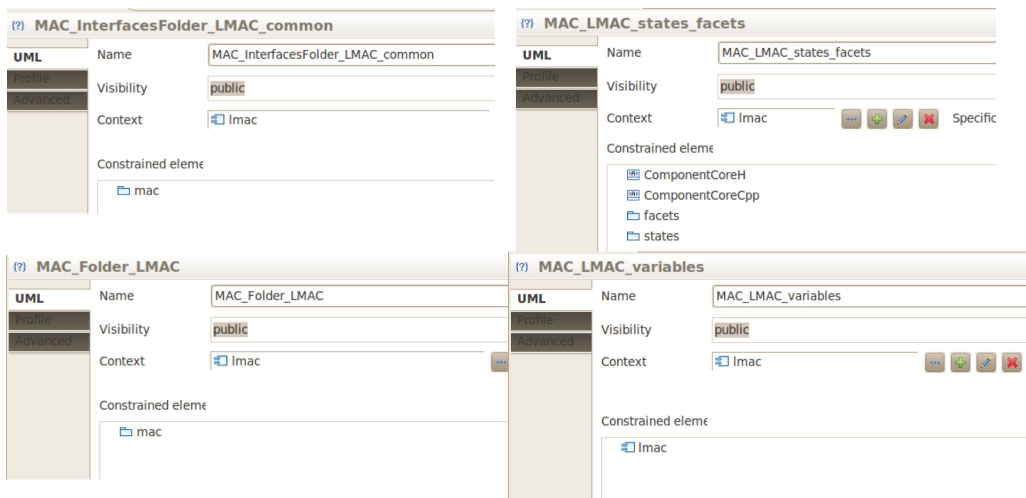

Figure 45 Constraints with different constrained elements

4.5.1. System Components

The model variables that may be associated with a component constraint are:

1. language

2. path
3. CFLAG
4. LFLAG
5. cc_compiler
6. cxx_compiler
7. path_compiler

4.5.2. Language

The variable $language defines the coding language of the complete application.

4.5.3. Path

As was mentioned previously, at least the *$path* variable has to be defined in the model. This variable has to be associated with the *System* component included in the *ApplicationView*. Through this variable, the designer annotates the absolute path where the application functionality files are allocated (Figure 46), which act as base paths for the rest of the system. This is mandatory.

4.5.4. CFLAGS and LFLAGS

The model variables associated with the *System* component of the *ApplicationView* can include the set of CFLAGs and LFLAGS required for the native compilation of the application (Figure 46).

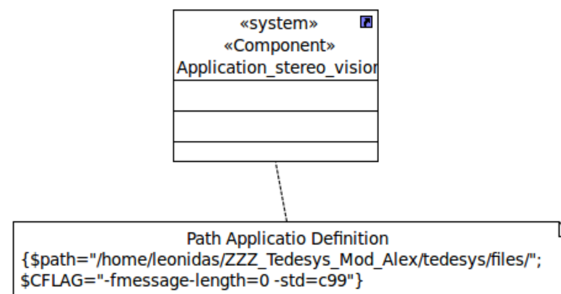

Figure 46 $CFLAGs for native compilation

4.5.5. Compiler and Compiler path

The model variables associated with the *System* component of the *ApplicationView* can include the compiler (for C or C++) required for native compilation and the path where this compiler is allocated (Figure 47). By default, gcc and g++ are the compilers considered for compilation.

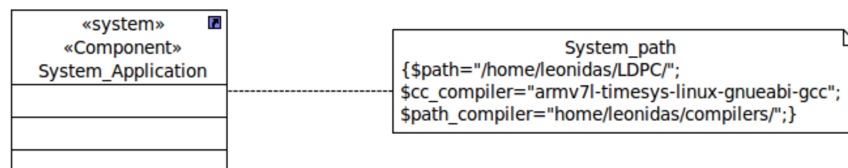

Figure 47 Compiler variable

4.6. Application Components

The model variables that can be associated with a *RtUnit* application component constraint are:

1. language
2. path
3. path_system
4. creation
5. CFLAG
6. LFLAG

4.7. 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.

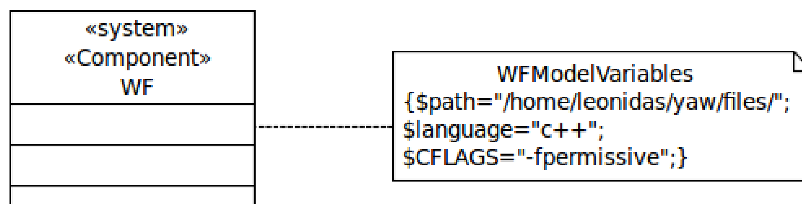

Figure 48 Specification of the System's base path

The base path is the *$path* annotated in the *System* component. This path is used for creating the complete paths of the different files, filesfolder, etc. of the application (Figure 48).

Then, each application component has its own relative path. In Figure 49, 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.

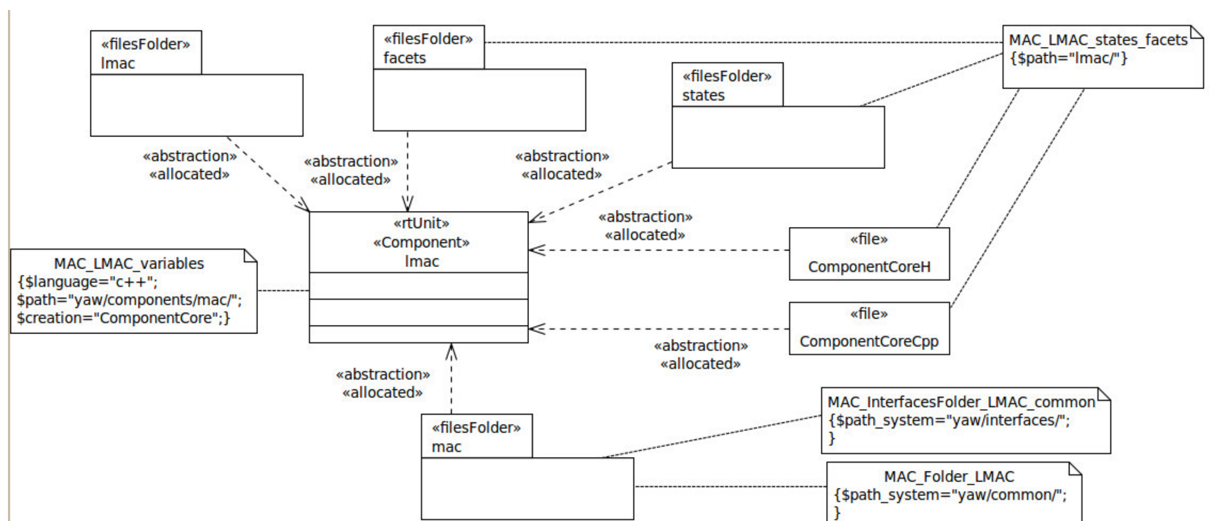

Figure 49 Application components with different types of model variables

To complete the path of the files "ComponentCoreH" and "ComponentCoreCpp" in Figure 49, 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 2.1) is concatened. 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 48) plus the application component path (Figure 49) and the name of the *FileFolder* (or *File*): "home/leonidas/yaw/files/components/mac/lmac/"

A diferent case is the specification of the path for the path "mac". This path has an associated constraint where a *$path_system* variable is annotated. In this, the creation of the path does not consider the base path of the application component (in Figure 49, "yaw/components/files/"). In this case, the *System* path (Figure 48) is concatenated with the value of the *$path_system* variable and the name of the *FilesFolder*: "home/leonidas/yaw/files/yaw/interfaces/mac/" and "home/leonidas/yaw/files/yaw/common/mac/".

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 (Figure 49, "mac" *FilesFolder*).

5. Concurrency View

In this view, the static threads of the system are defined. The static threads are created when the aplication components associated with them are triggered. Aditionally, this view includes the mapping of the application elements defined in the *ApplicationView* on these threads.

The UML elements used in this view are:

1. UML Component for modeling the thread types and other Components in order to define thread instances and the mapping of application elements onto these threads.
2. UML Abstraction for associating application instances to threads
3. UML generalization for relating the *System* component of *ApplicationView* with the *System* component of this view.

Class diagrams are used for defining the thread types and for capturing the UML generalization of the *System* components of *ApplicationView* and *ConcurrencyView*.

Composite structure diagrams are used for defining the thread instances and the association application-thread.

5.1. Thread modeling

The threads are modelled as UML components specified by the MARTE stereotype <<SwSchedulableResource>>. The threads execute a function, so each thread requires an associated function. These thread functions are modeled as UML operations attached to the *SwSchedulableResource* component. There are two different cases for modelling the function threads:

A *SwScheduleResource* includes a number of functions to be executed concurrently. The number of operations included in the *SwScheduleResource* is understood to be the number of threads to be created, since a thread only can execute one function (Figure 50).

Several *SwSchedulableResource*s includes only one UML operation (Figure 50).

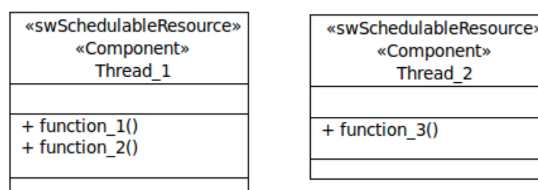

Figure 50 Thread components

5.2. Thread structure

The thread structure is captured in a System component included in the view. The threads structure is composed of instances of the *SwScheduleResource* components previously defined (Figure 51).

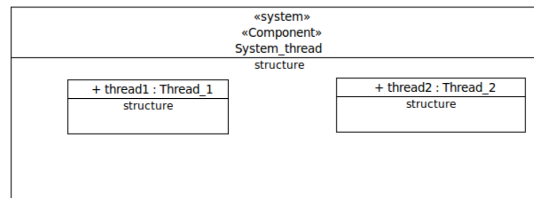

Figure 51 Thread instances which compose the thread structure

In this view, the association between application components and static threads associated with functions is established.

5.3. Application-Thread association

The *System* component of the *ConcurrencyView* is used in order to allocate the application instances defined in the *ApplicationView* within the corresponding threads. 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 *ConcurrencyView* package (Figure 56).

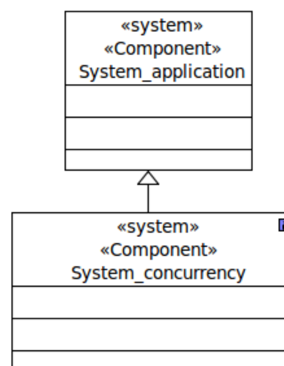

Figure 52 Generalization of the *System* component of the *Concurrency View*

By means of a UML composite structure diagram associated with the *System* component, the application instances defined in the *System* component of the *ApplicationView* within the theads can be mapped. The application component instances are associated with thread instances by means of UML abstractions specified by the MARTE stereotype <<allocate>> (Figure 53).

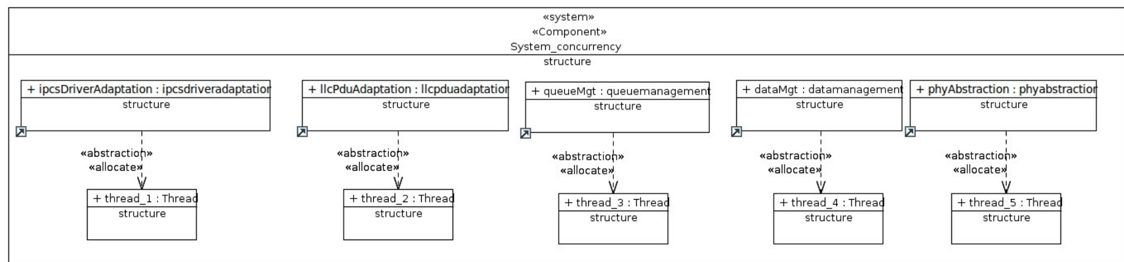

Figure 53 Application-thread association

5.4. Initial function values

In some cases, the functions executed by threads have associated specific, initial values to its parameters. For modelling that, a UML constraint should owned by the *System* component of the view. In this constraint, there are annotated the name of the functions and the values of their parameters by using the annotation "$initValue=nameFunction(value1,value2,value3)".

6. 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:

1. UML Component for modeling the memory partition types and other Components in order to define executables
2. UML Generalization for relating the *System* component of the *ApplicationView* with the *System* component of the *MemorySpaceView*.
3. 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.

6.1. Process modelling

Memory partitions are modeled by the MARTE stereotype <<MemoryPartition>> applied on a UML component (Figure 54).

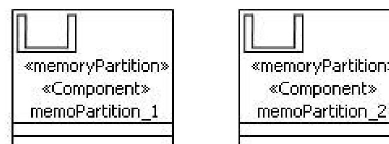

Figure 54 Memory partitions

6.2. Process structure

The executables are defined in a *System* component included in the view as instances of the *MemoryPartition* components previously defined (Figure 55).

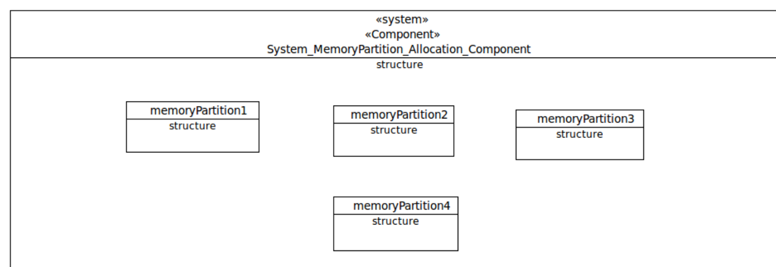

Figure 55 Executables definition

6.3. Application Allocation structure

In this view, the allocaction of the application components to the memory partitions (executables) is dealt with.

This *System* component is used in order to allocate the application instances defined in the *ApplicationView* to the corresponding memory paritions. 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 56).

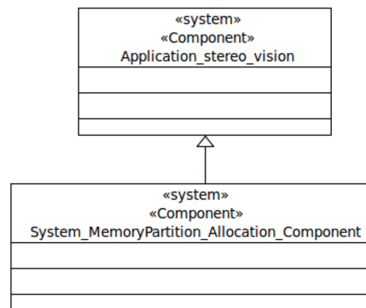

Figure 56 Specialization of the *System* component of Memory Allocation View

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>>.

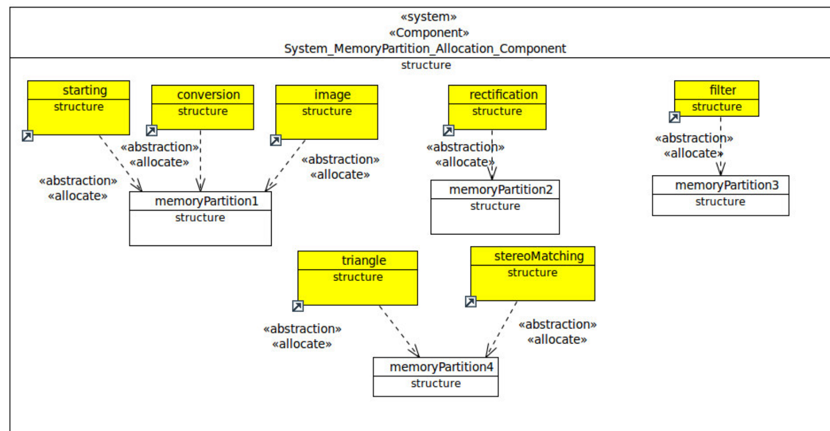

Figure 57 Memory partition allocation

In Figure 57, the yellow boxes are application components that are mapped onto memory partitions.

PDM Views

7. HW Resources View

The *HwResourceView* includes all the HW components required for the specification of the platform architecture. These HW components act as a palette; using instances of these HW components, the designer specifies the HW architecture structure defined in the *ArchitecturalView*.

The UML elements used in this view are:

4. UML Components for modeling the HW component types

Class diagrams are used for defining the HW components.

The MARTE stereotypes used to specify the HW components that can be captured in the *HwResourcesView* are shown below.

UML2 Diagram elements	MARTE profiles	MARTE stereotypes
Component	HRM	HwProcessor HwRAM HwROM HwCache HwDMA HwBus HwMedia HwEndPoint HwBridge HwI_O HwPLD HwASIC HwDevice HwSensor HwISA

Table 4 MARTE stereotypes used for refining the HW platform

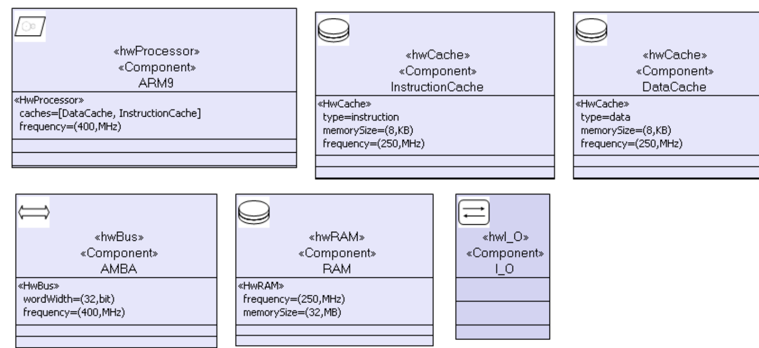

Figure 58 HW platform resources

7.1. HW Processors

The HW processors are modelled by the MARTE stereotype <<HwProcessor>>.

7.2. Processor ISA

The HwProcessor can be more specifically defined by an 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 mandatory for the PHARAON methodology:

family: NFP_String. Defines the ISA family

type: ISA_Type. Specifies the ISA type.

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.

In the case of this modeling methodology, the possible values of the family attribute are DSP, GPU, CortexA, undef.

7.2.1. DSP processors

This value denotes that the processor is a DSP (Digital Signal Processor). The Eclipse plug-in generates the entire code infrastructure to execute an application component in this HW resource.

7.2.2. GPU processors

This value denotes that the processor is a GPU (Graphical Processing Unit). The Eclipse plug-in generates the entire code infrastructure to execute functions in this HW resource.

7.2.3. CPU co-processors

CPUs may have associated co-processors which may affect the compilation process. So, the "CortexA" processor has an associated NEON co-processor (www.arm.com/products/processors/technologies/neon.php). In the case that a *HwProcessor* has an associated *HwISA* specified as "CortexA?" (where the "?" represents any possible value, Figure 59), the eclipse plug-in generates the entire infrastructure for using the NEON co-processor to execute functionality. The designer can select which application components should be executed in the NEON co-processor.

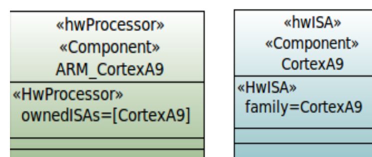

Figure 59 HW Specification of a CortexA processor

7.3. Processor Caches

Each *HwProcessor* can have associated cache memories. The caches can be associated with a HwProcessor by means of the attribute *caches* of the stereotype *HwProcessor*. This stereotype attribute selects the UML components that are characterized by *HwCaches*.

7.4. HW Processor variables

Some additional model variables have to be defined for specifying some required platform characteristics. These variables are used for specifying the C and C++ compilers and the different LFLAGs and CFLAGs in order to implement the make files for the system cross compilation. These variables are:

- **$cc_compiler:** defines the name of the cross compiler for C.
- **$cxx_compiler:** defines the name of the cross compiler for C++.
- **$path_compiler:** defines the path where the cross compiler is allocated.
- **$CFLAG:** defines the compilation flags for the cross compilation.
- **$LFLAG:** defines the linking flags for the cross compilation.

These variables are specified in a UML constraint (Figure 60). This constraint is owned by the HW Processor (the attribute "Context" has to contain the HWProcessor

component to be constrained) and associated with a HwProcessor component by uisng a UML link.

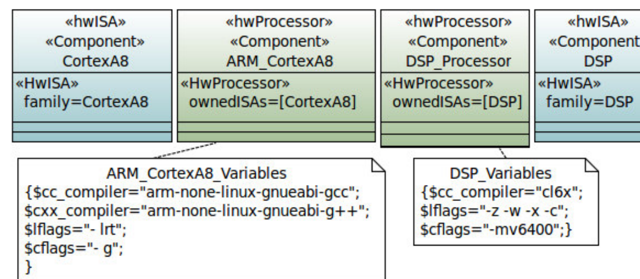

Figure 60 HwProcessor compilers

7.5. Network

A network is modelled by using the MARTE stereotype *HwMedia*.

7.6. Network Interfaces

The network interfaces are modelled by the MARTE stereotype <<HwEndPoint>>. In the PHARAON methodology, each *HwEndPoint* component should have an attribute called *IPAddress.* In the attribute, *Default Value* specifies the IP address by using an UML Literal String, in order to denote the IP address to enable the TCP/IP communication. This *IPAddress* should be different for ach *HwEndPoint* component. As a modelling constraint, only one instance of *HwEndPoint* component can be included in an execution node.

7.7. I/O Components

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

7.8. HW components' Functional Modes

The HW components can have different associated functional modes that specify different characteristics that define the HW component's behaviour 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>>.

For characterizing the functional attributes previously mentioned, some modelling elements have been used. The first one is taken from the paper[1], specifically the stereotype <<HwPowerState>>, in order to specify the *frequency* of the HW component 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>>.

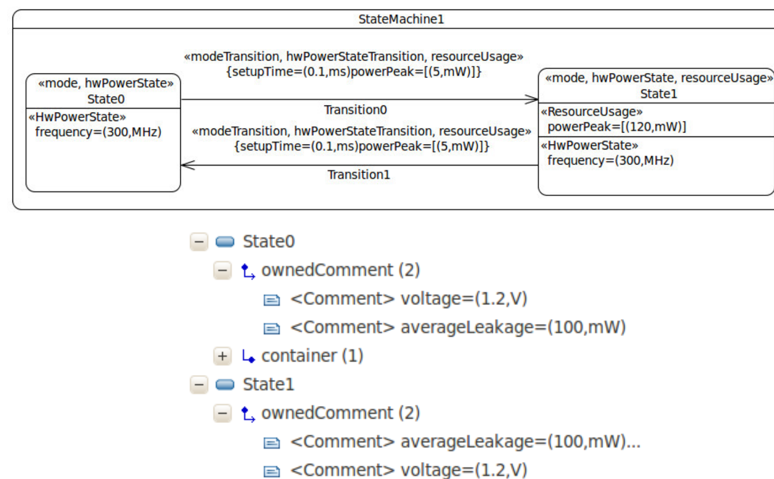

Figure 61 HwProcessor mode specification

[1] T. Arpinen, E. Salminen, T.D. Hämäläinen, M. Hänniikäinen. "MARTE profile extension for modeling dynamic power management of embedded systems". JSA, April 2012, Pages 209–219.

8. 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]
policy: SchedulingPolicyKind[1];
drivers: DeviceBroker [*]
interProcessCommunication: [1]

Figure 62 OS stereotype attributes

The type of the OS is defined in the *type* attribute (linux, windows, etc.). The attribute *policy* defines the scheduling policy of the *OS* component.
The possible values of the SchedulingPolicyKind are:

- *Undef:* modeling the default policy of Linux systems
- *FixedPriorityPre-emptive*: a fixed priority rank to every process is assigned, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.
- *RoundRobin*: the scheduler assigns a fixed time unit per process, and cycles through them.
- *FIFO*: First In First Out simply queues processes in the order that they arrive in the ready queue.
- *EarliestDeadlineFirst*: whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline.
- *RateMonotonic*: static priorities are assigned on the bases of the cycle duration of the job: the shorter the cycle duration is, the higher is the job's priority.
- *LeastLaxityFirst*: it assigns priority based on the *slack time* of a process. Slack time the time a job would take to finish if the job was started now. It imposes the simple constraint that each process on each available processor possesses the same run time, and that individual processes do not have an affinity to a certain processor.
- *Lottery*: processes are each assigned a random number, and the scheduler chooses one at random to select the next process.
- *TableDriven*: the scheduler applies a predefined fixed repetitive schedule.

- *ShortestJobFirst*: the scheduler arranges processes with the least estimated processing time remaining to be next in the queue.

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.

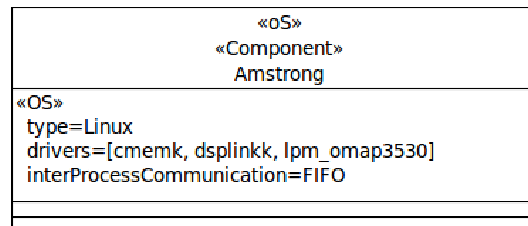

Figure 63 OS component

8.1. 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

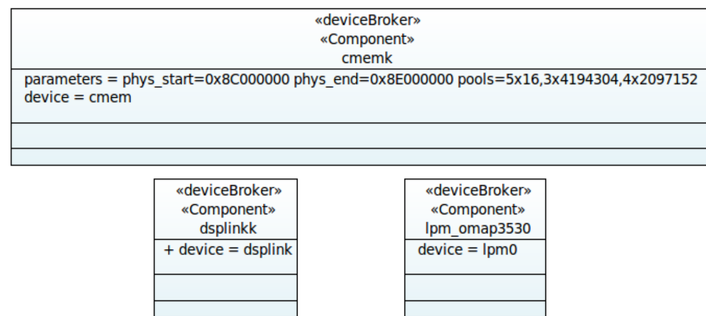

Figure 64 Driver for DSP management

8.2. Repository

The "repository" property denotes the url direction 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.

8.3. Parameters

The "parameters" property denotes the set of paramaters 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.

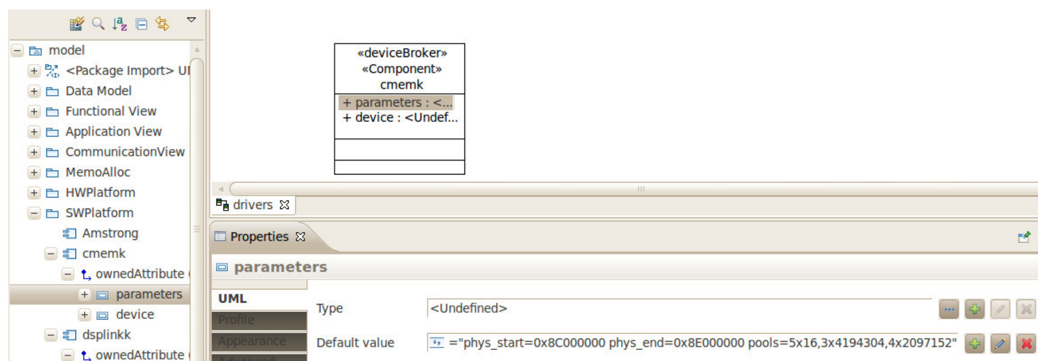

Figure 65 "Parameter" driver property

8.4. 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.

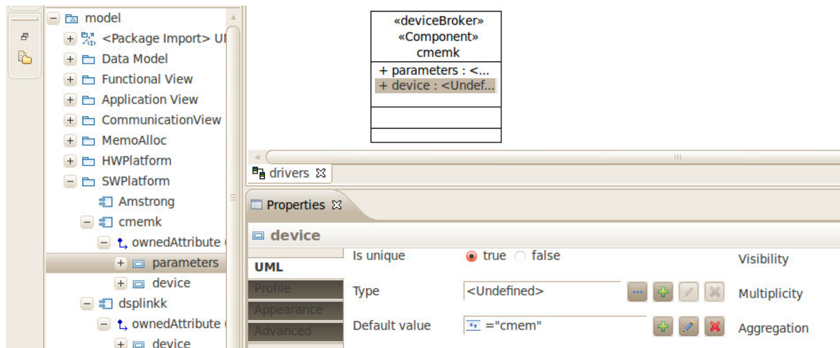

Figure 66 “Device” driver property.

PSM Views

9. Architectural View

The *Architectural view* focuses on the architectural aspects of the system related to the instantiation and assembly of HW and SW platform components. The *Architectural view* is displayed using a diagram where the following items are described:

- Definition of the SW platform (e.g. OSs).
- Definition of the HW resources (processors, memories, buses, network, etc.).
- Interconnection of these HW resources
- Association of the HW resources with OSs.

The UML elements used in this view are:

5. UML Component for modeling HW/SW architecture structure
6. UML Abstraction for associating OSs with computing resources

Composite structure diagrams enable the HW/SW architecture structure and the different mappings onto the HW/SW resources.

9.1. Modelling of the HW/SW platform architecture

The *Architectural View* contains a component specified by the stereotype <<System>>. Through a composite structure diagram associated with this *System* component, the instances of HW components defined in the *HW Resources View* are specified and interconnected in order to create the HW architecture platform. These interconnections are carried out through UML ports connected by UML connectors (Figure 67).

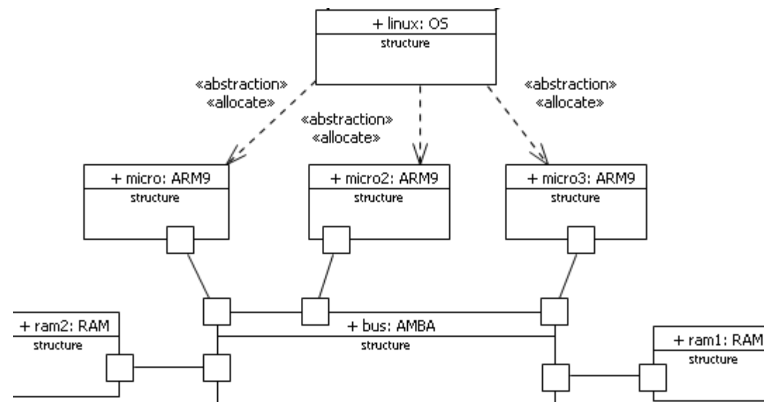

Figure 67 HW & SW platform architectures

The *System* component also includes the definition of the SW platform architecture. The SW platform architecture is composed of instances of the *OS* components included in the *SWPlatformView*.

Only one *System* component should be defined within the *Architectural View* package.

9.2. Allocation of SW instances to HW instances

The association of the *OS* instances with HW resources instances is modelled by means of UML abstraction specified by the MARTE stereotype <<allocate>> (Figure 67).

9.3. Architectural Allocation

This *System* component should be specialized by the *System* component defined in the *MemorySpaceView* and the *System* component defined in the *ConcurrencyView*. This specialization is modelled by means of a UML generalization defined in a UML class diagram. In this way, this new *System* component can refer to the *MemoryPartition* instances defined in the *MemorySpaceView* and the thread instances defined in the ConcurrencyView.

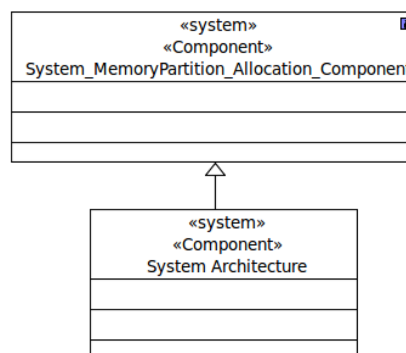

Figure 68 Specialization of the *System* component of *Architectural View*

The *MemoryPartition* instances are allocated to SW resource instances. The assignation of a *MemoryPartition* instance to a SW resource instance is done through a UML abstraction where the MARTE stereotype <<Allocate>> has been applied.

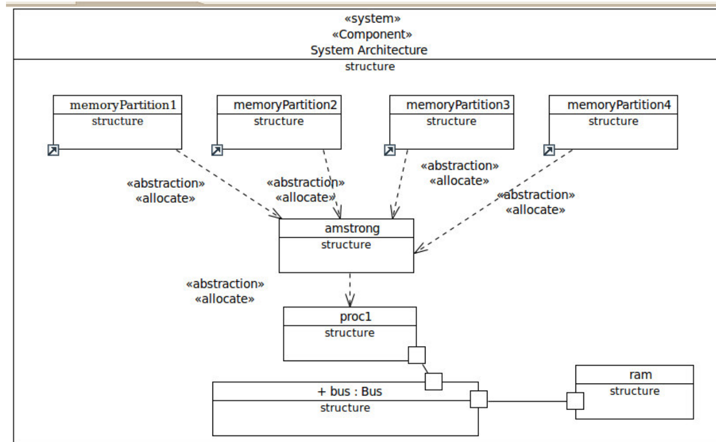

Figure 69 Memory partition allocation on HW/SW platform

9.4. Allocation on DSP

When the the memory allocation is done on a DSP, the allocation is captured by means of a UML abstraction specified by the MARTE stereotype <<Allocate>>. However, the mapping is captured directly from *MemoryPartition* instance to the DSP resource, without any OS in the middle (Figure 70).

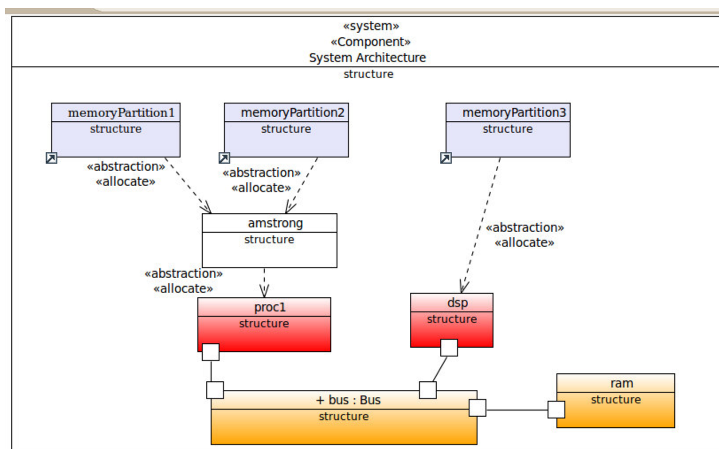

Figure 70 Memory partition allocations to DSP

The memory partition instance mapped onto the DSP HW resource has a modelling restriction, only one application component can be allocated to a memory partition that is mapped onto a DSP (Figure 70 and Figure 71).

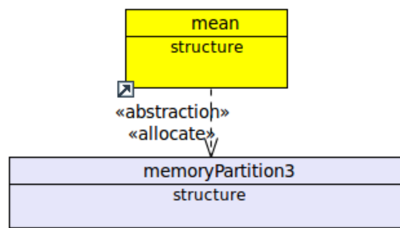

Figure 71 Application component allocation to a memory partition

9.5. Multiple HW resources allocation

The modellining methodology enables multiple allocations of the memory spaces in different HW resources of the platform as ca ben seen in Figure 72.

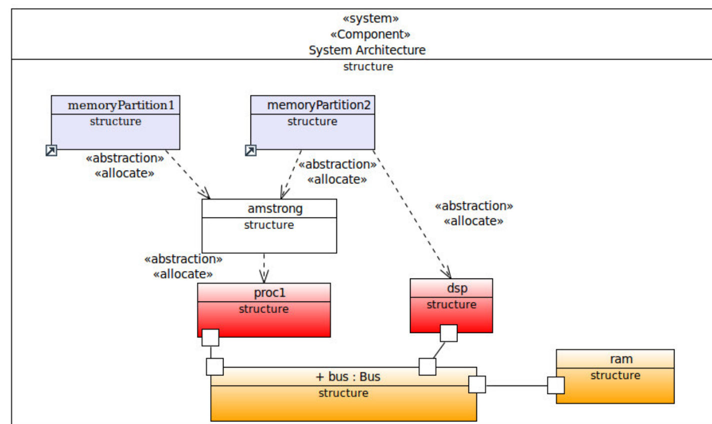

Figure 72 Multi HwResources allocation

9.6. Application Allocation to GPU

The application components are mapped onto memory partitions and then, these memory partitions are mapped onto HW/SW resources of the platform. A special case of application mapping is the mapping onto GPU HW resources. The mapping onto these GPU HW resources requires the mapping of functions of the application component. In order to enable this mapping, the functions to be executed in the GPU resource have to be specifed in the model.

For this purpose, UML OpaqueBehaviours should be included in the application component specification. These OpaqueBehaviours have to be named with the corresponding functions that they represent (Figure 73).

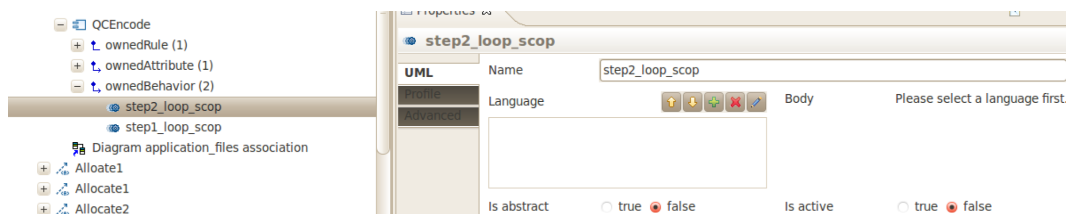

Figure 73 Application functions for GPU mapping

The GPU instance has to be presented in another UML composite structure diagram of the *System* component,. Additionally, in this new diagram, the corresponding instances of functions should be presented. These functions are captured as UML OpaqueBehaviours and they are owned by the particular application component. Instances of these *OpaqueBehaviour* functions are included in the composite structure diagram. Then, these instances are allocated to the GPU resource by using UML abstraction specifed by the MARTE stereotype <<allocate>>.

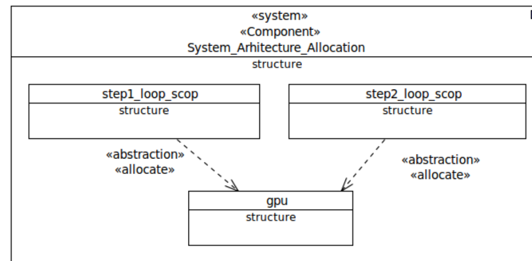

Figure 74 Function-GPU allocation

9.7. Thread allocation

The threads can be allocated to processors in order to define the thread affinity and balance the system for the processing load distribution. For this purpose, the thread instances defined in the ConcurrencyView are mapped onto the processors by using UML abstraction specifed by the MARTE stereotype <<allocate>> (Figure 75).

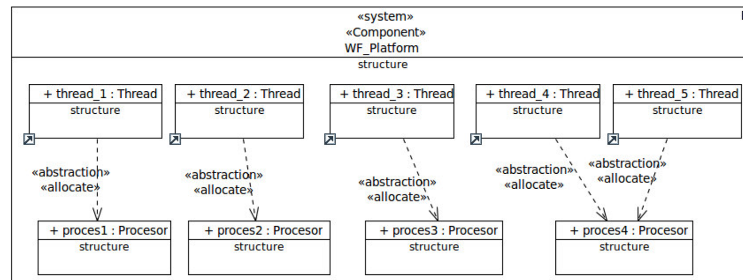

Figure 75 Thread-processor mapping.

9.8. Processor identifier

In some cases, specifically for defining the affinity of a thread, an identifier should label the processor instances of the platform. For that purpose, in the attribute “Default Value” of the processor instance, associate a LiteralInteger. In this element, the identifier is annotated.

10. 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.

For modeling the environment, a set of stereotypes of the UML standard profile UTP has been selected.

10.1. 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>> (Figure 76).

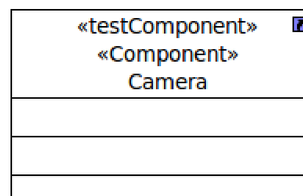

Figure 76 Environment component

10.2. 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>> (Figure 77). The environment application components should be included in the *ApplicationView* like the rest of the application components of the system.

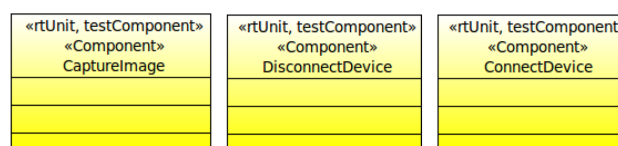

Figure 77 Environment application components

All these *RtUnit-TestComponent* components can have the same associated modeling elements (threads, file folder, files) as the rest of the application components.

These *RtUnit-TestComponent* application components have associated C files. These 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 78).

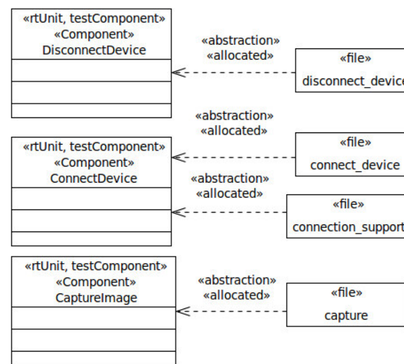

Figure 78 Environment Application components with associated Files

10.3. 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 intances of *RtUnit-TestComponent* application components (Figure 79) in a Composite structure diagram associated with the environment *TestComponent* component.

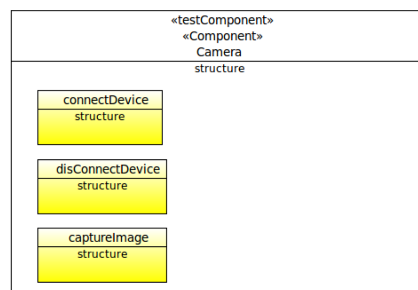

Figure 79 Application instances of an environment component

10.4. Environment component structure: ports

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

The ports that have been specifed by the ClientServerPort stereotype are those of the environment component (*TestComponent* component), as can be seen in Figure 80 (Camera *TestComponent*). These *TestComponent* ports are connected to the internal application instance ports by using UML connectors (Figure 80). These application

instance ports have to be named similarly to the *TestComponent* port that they are connected to (Figure 80).

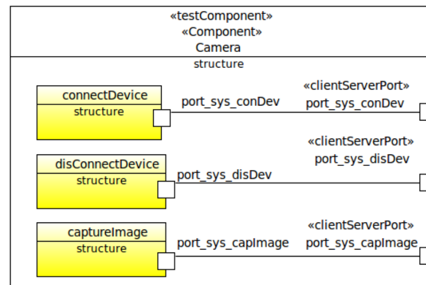

Figure 80 Environment Application components

10.5. Environment structure

The environment structure is composed of insances of environment components connected to the *System*.

The environment structure is modelled in a UML component specifed 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* (Sytem Under Test).

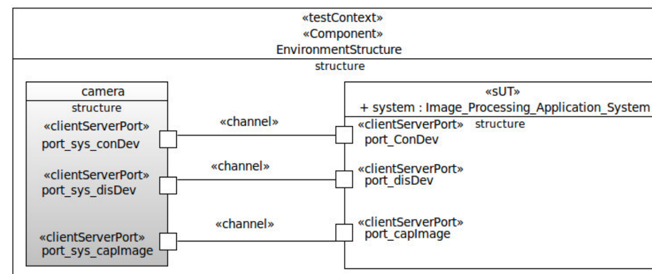

Figure 81 Definition of the environment structure

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

10.6. Memory allocation

The Environment elements have to 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 modelled by means of a UML generalization defined in a UML class diagram (Figure 82).

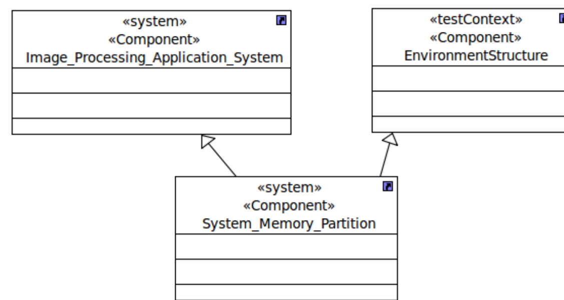

Figure 82 Generalization of Environment structure with the *System* component of the *MemorySpaceView*

Then, the allocation on memory spaces of the environment component (instances of *TestComponent* components) can be done (Figure 83).

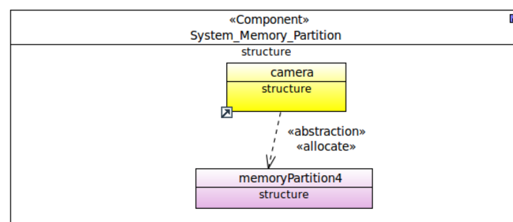

Figure 83 Allocation of environment component to the memory partitions

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.

Annex

1. Methodology Stereotypes

Stereotype	Attributes	Profile
DataView		PHARAON
FunctionalView		PHARAON
ConcurrencyView		PHARAON
ApplicationView		PHARAON
MemorySpaceView		PHARAON
HWResourcesView		PHARAON
SwResourcesView		PHARAON
ArchitecturalView		PHARAON
VerificationView		PHARAON
Tupletype		MARTE
CollectionType	collectionAttrib:property [0..1]	MARTE
DataSpecification	size:NFP_Data [1] pointer:Boolean [1] dataSpecifier: Specifier [1] dataQualifier: Qualifier [1] complexDataType : String [0..1]	PHARAON
File		Standard UML
ApplicationFile	parallelized: Boolean [1] highLevel: Boolean[1] implementation: String [0..1]	PHARAON

	notModifiable: Boolean [1] environment: Boolean [1]	
SystemFile	systemConfiguration: Boolean [1] systemMetrics:Boolean[1] environment: Boolean [1] RTL: Boolean [1] TLM: Boolean [1]	PHARAON
FilesFolder	parallelized: Boolean [1] highLevel: Boolean[1] implementation: String [0..1] notModifiable: Boolean [1] environment: Boolean [1]	PHARAON
ClientServerSpecification		MARTE
Pointer		PHARAON
CommunicationMedia		MARTE
StorageResource	result : NFP_Integer[0..1]	MARTE
ChannelTypeSpecification	blockingFunctionDispatching:Boolean [1] blockingFunctionReturn:Boolean [1] priority : integer [0..1] timeOut:NFP_Duration [0..1] ordering: Boolean [1]	PHARAON
NotificationResource		MARTE
SharedDataComResource	identifierElements: TypedElement= [0..*]	MARTE
RtUnit	isMain : Boolean main : Operation [0..*] srPoolSize: Integer [0..1] srPoolPolicy : PoolMgtPolicyKind [1]	MARTE

SwSchedulableResource		MARTE
create		UML standard
Allocated		MARTE
ClientServerPort	kind : ClientServerKind [1] provInterface : Interface [0..1] reqInterface : Interface [0..1]	MARTE
Channel	commType: ChannelTypeSpecification [1] communicationEngine: CommunicationEngineKind[1] communicationOSService: communicationOSServiceKind [1]	PHARAON
System		PHARAON
MemoryPartition		MARTE
Allocate		MARTE
HwProcessor	ownedISA : HwISA [0...1] caches : HwCaches[*]	MARTE
HwRAM		MARTE
HwROM		MARTE
HwCache	type : CacheType [1] level: NFP_Natural [0..1]	MARTE
HwDMA		MARTE
HwBus		MARTE
HwMedia		MARTE
HwEndPoint		MARTE
HwBridge		MARTE
HwI_O		MARTE
HwPLD		MARTE

HwASIC		MARTE
HwDevice		MARTE
HwSensor		MARTE
HwISA	family: NFP_String [0..1] type: ISA_Type	MARTE
DseProcessorParameter	name: String[1] parameter: ProcessorParameter [1] unit: Units [1] max: String [0..1] min: String [0..1] step: String [0..1] items: String [*]	PHARAON
Mode		MARTE
HwPowerState	frequency : NFP_Frequency [0..1]	MARTE
ModeTransition		MARTE
HwPowerState Transition	setUp : NFP_Duration [0..1]	MARTE
ResourceUsage	powerPeak : NFP_Power [0..1]	MARTE
OS	type:String [1] policy: SchedulingPolicyKind[1] drivers: DeviceBroker [*] interProcessCommunication: InterProcessCommunicationMechanism [1]	PHARAON
DeviceBroker		MARTE

TestComponent		UTP
TestContext		UTP
SUT		UTP
Refine		UML Standard
Reference		PHARAON
Qualifier	qualifier:Qualifier [1]	PHARAON

Table 5 List of Stereotyes and attributes used in PHARAON methodology.

2. Methodology Enumerations

Enumeration	Values	Profile
Specifier	None Char signed char unsigned char short short int signed short signed short int unsigned short unsigned short int int signed int unsigned unsigned int long long int signed long signed long int unsigned long	PHARAON

	unsigned long int long long long long int signed long long signed long long int unsigned long long unsigned long long int float double long double void	
Qualifier	None Const Volatile register	PHARAON
PollMgtPolicyKind	infiniteWait timedWait dynamic exception other	MARTE
ClientServerKind	proreq provided required	MARTE
CacheType	data instruction unified	MARTE
ISA_Type	RISC CISC VLIW SIMD	MARTE

	Other Undef	
ProcessorParameter	frequency voltage	PHARAON
SchedulingPolicyKind	Undef FixedPriorityPre-emptive RoundRobin FIFO EarliestDeadlineFirst LeastLaxityFirst Lottery TableDriven ShortestJobFirst	MARTE/PHARAON
CommunicationEngineKind	undef default MCAPI OPenMP OpenStream TCP/IP	PHARAON
CommunicationOSServiceKind	undef FIFO Socket messgeQueue SharedMemory File	PHARAON
InterProcessCommunicationMechanism	FIFO Socket MessageQueue SharedMemory File	PHARAON