UML/MARTE Methodology for Heterogeneous System design

May, 2016

Microelectronics Engineering Group

TEISA Dpt. , University of Cantabria

Authors: P. Peñil

Revisions : F. Herrera

Author	Date	Description
FeHe	26/05/2015	Polishing of the Architectural View section
FeHe	28/05/2015	Polishing on the Application View section including some restructuring
FeHe	25/06/2015	Polishing of section 5.3
FeHe	15/05/2016	Polishing of section 5.1

Index:

Index of Tables:

Index of Figures:

1 Model View specification

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>>

<<MemorySpaceView>>

<<HwResourceView>>

<<SwPlatformView>>

<<ArchitecturalView>>

<<VerificationView>>

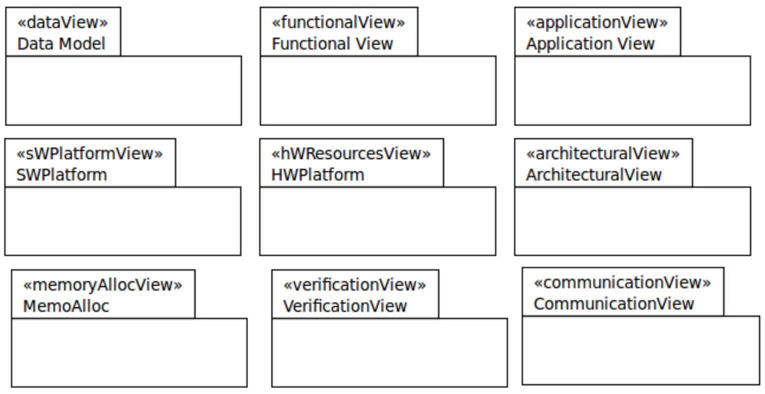


Figure 1 Model views

PIM Views

2 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.

2.1 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 2).

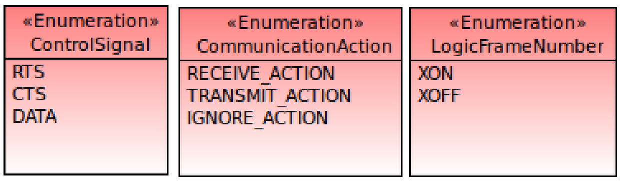


Figure 2 Enumeration data types

2.2 Primitive Data type

The UML PrimitiveTypes are used to define basic data types. As can be seen in Figure 3, all these data definitions are classic primitive data types in coding.

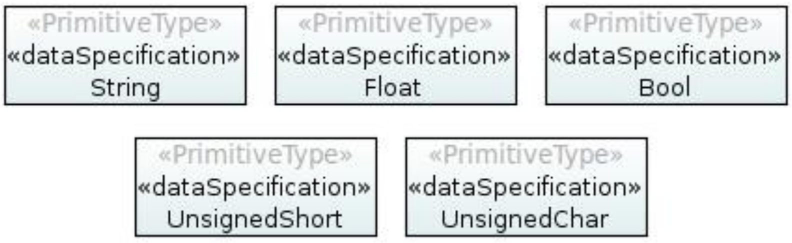


Figure 3 Primitive types

2.3 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.

2.3.1 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.

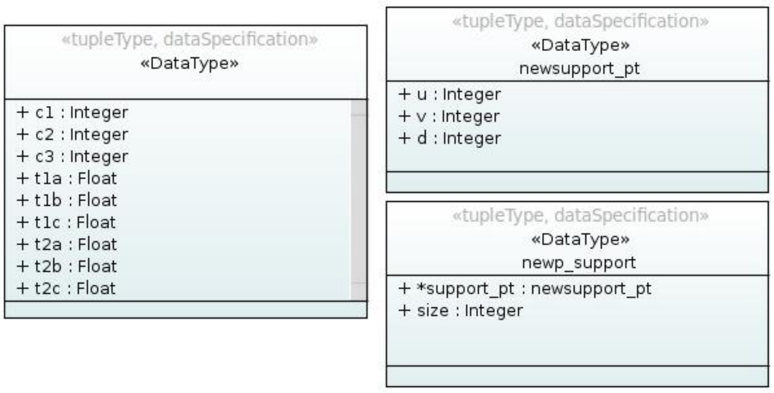


Figure 4 Structure Datatype

When a field of the structure data types is a pointer, an asterisk is annotated in the name ("newp_support" data type of Figure 4).

2.3.2 Array Data type

Arrays are modelled by using the MARTE stereotype <<CollectionType>>. The *collectionType* stereotype is applied to a *DataType* model element. A property has to be added to this DataType. The property should be typed by PrimitiveType or another DataType. Then, in the attribute *collectionAttrib* of the stereotype *CollectionType* that property should be attached (in Figure 5, property "array128i").

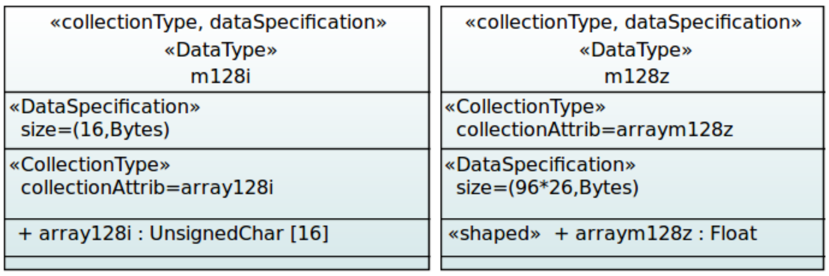
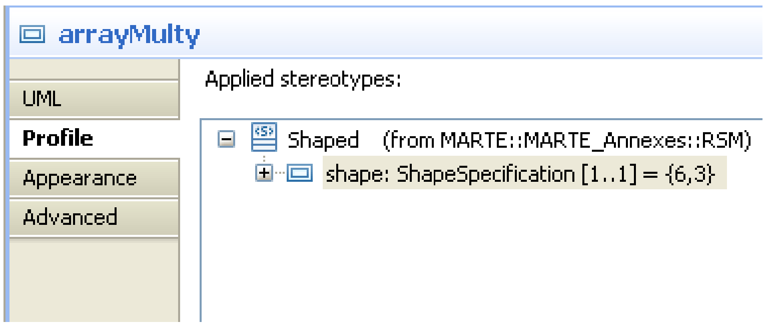

«collectionType, dataSpecification»
«DataType»
m128i
«DataSpecification»
size=(16,Bytes)
«CollectionType»
collectionAttrib=array128i
+ array128i : UnsignedChar [16]
«collectionType, dataSpecification»
«DataType»
m128z
«CollectionType»
collectionAttrib=arraym128z
«DataSpecification»
size=(96*26,Bytes)
«shaped» + arraym128z : Float

Figure 5 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 5 and Figure 6). 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 5).



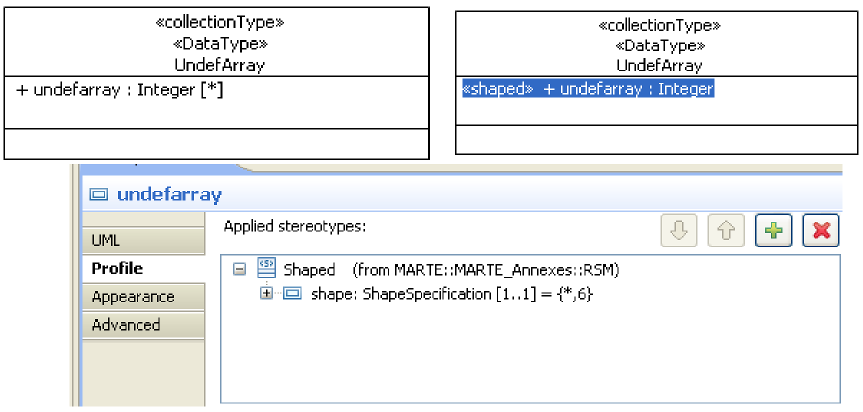
Applied stereotypes:

Shaped (from MARTE::MARTE_Annexes::RSM)

shape: ShapeSpecification [1..1] = {6,3}

Figure 6 Array dimension specification by the Shape stereotype

In some cases, the designer can define the dimensions of an array with no specific value. Figure 7 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 7 shows these annotations.



«collectionType»
«DataType»
UndefArray

+ undefarray : Integer [*]

«collectionType»
«DataType»
UndefArray

«shaped» + undefarray : Integer

Applied stereotypes:

Shaped (from MARTE::MARTE_Annexes::RSM)

shape: ShapeSpecification [1..1] = {*,6}

Figure 7 Undef dimesion of an array

2.4 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 8 <<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

2.5 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 9, 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 9, the data "Byte") the specific data is specified by the DataType (in Figure 9 the "QoS" is specified as "BYTE").

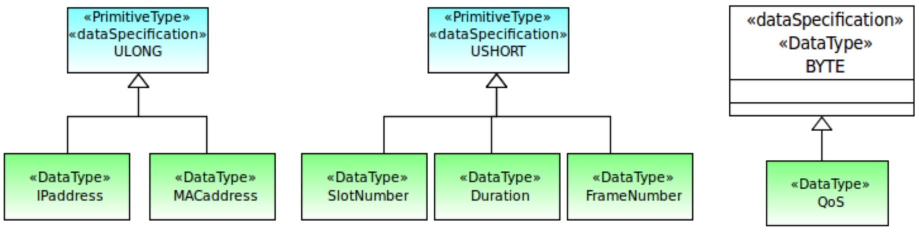


Figure 9 Data Generalizations

2.5.1 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 10, the Data Type *DataType_Exploration*) should be specifed by the name of the general element of the UML generalization (in Figure 10, 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.

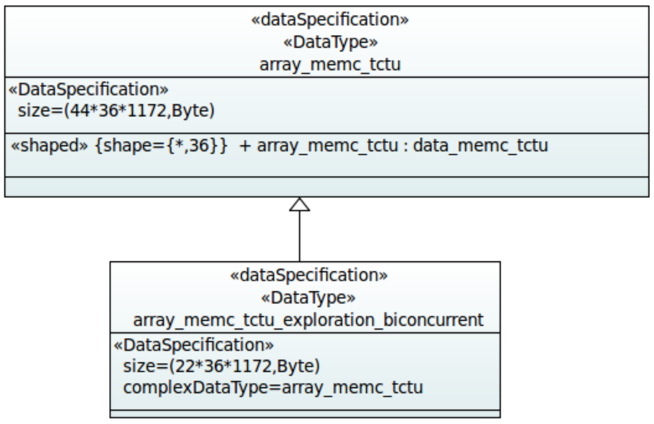


Figure 10 Data Type generalization for Concurrency exploration

3 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.

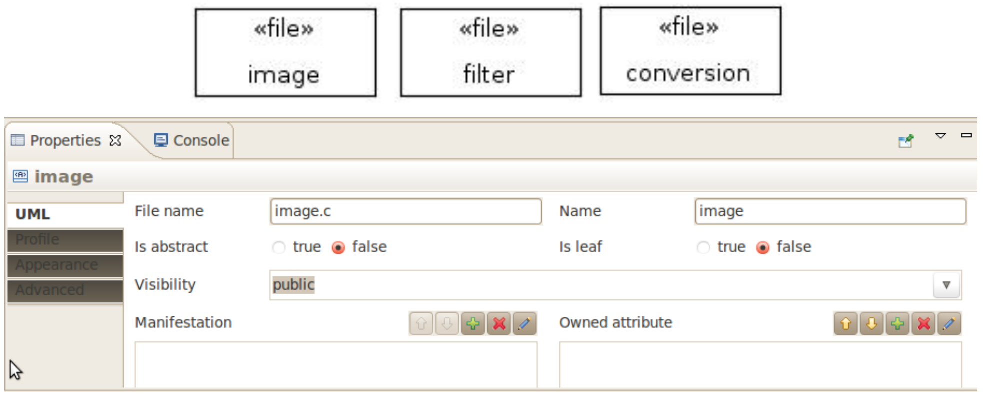
The UML elements used in this view are:

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.

3.1 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 11).



File name	image.c	Name	image
Is abstract	true [✓ false]	Is leaf	true [✓ false]
Visibility	public		
Manifestation		Owned attribute	

Figure 11 Files

3.2 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**: Boolen. 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 12 ApplicationFile stereotype attributes

3.3 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.

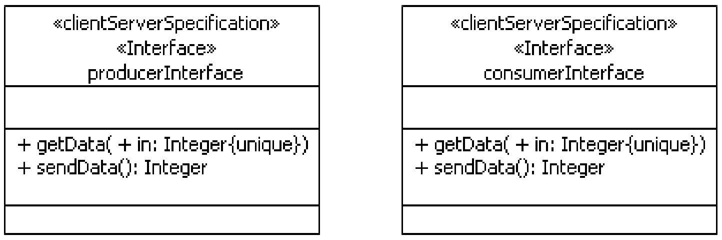

«clientServerSpecification»
«Interface»
producerInterface
+ getData(+ in: Integer{unique})
+ sendData(): Integer
«clientServerSpecification»
«Interface»
consumerInterface
+ getData(+ in: Integer{unique})
+ sendData(): Integer

Figure 13 Interfaces

3.3.1 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

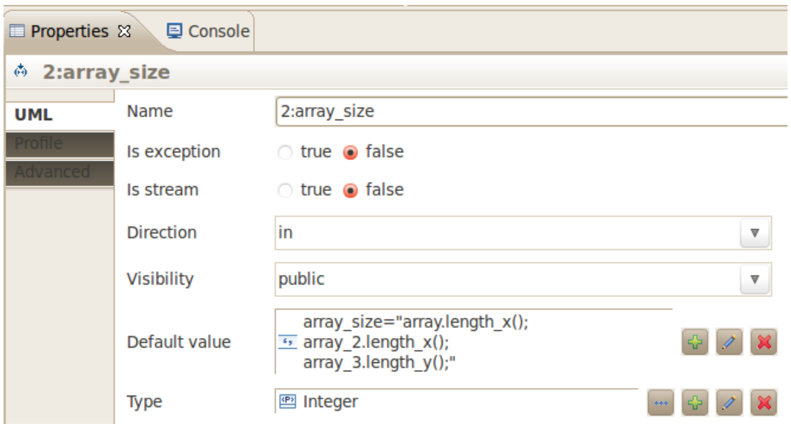
The function arguments can be specifed by a qualifier by appliying the stereotype <<ParameterQualifier>>. Values associated with the *ParameterQualifier* stereotype are “const”, “volatile” and “register”.

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:

- 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 14).



2:array_size

Name	2:array_size
Is exception	true [✓ false]
Is stream	true [✓ false]
Direction	in
Visibility	public
Default value	array_size="array.length_x(); array_2.length_x(); array_3.length_y();"
Type	Integer

Figure 14 Array size arguments

3.3.2 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.

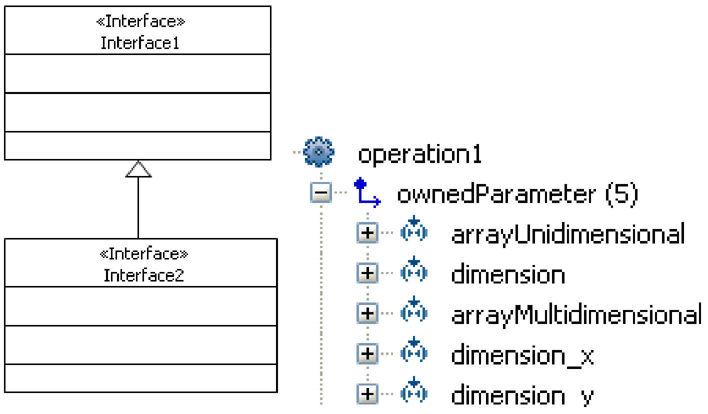


Figure 15 Interface generalization and operation1 of Interface1

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 16).

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 16).

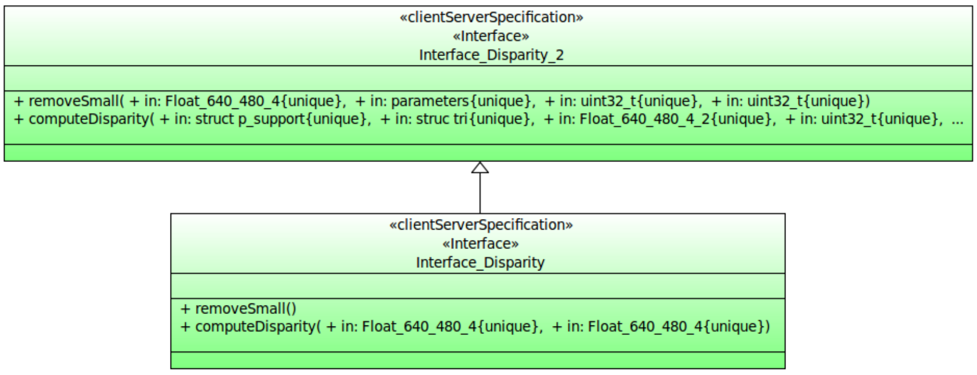

«clientServerSpecification»
«Interface»
Interface_Disparity_2
+ removeSmall(+ in: Float_640_480_4{unique}, + in: parameters{unique}, + in: uint32_t{unique}, + in: uint32_t{unique})
+ computeDisparity(+ in: struct p_support{unique}, + in: struc tri{unique}, + in: Float_640_480_4_2{unique}, + in: uint32_t{unique}, ...
«clientServerSpecification»
«Interface»
Interface_Disparity
+ removeSmall()
+ computeDisparity(+ in: Float_640_480_4{unique}, + in: Float_640_480_4{unique})

Figure 16 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 17) are used. All these interfaces have the same function associated, “stereo_matching”. However, the declaration of this function in these interfaces is very 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 17, 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".

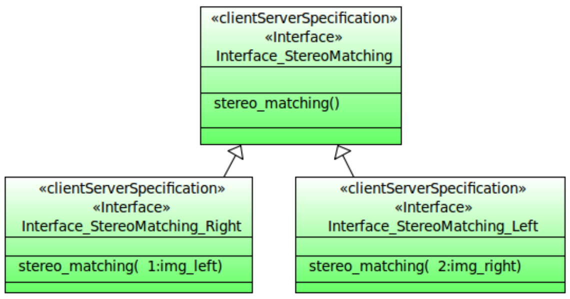

«clientServerSpecification»
«Interface»
Interface_StereoMatching
stereo_matching()
«clientServerSpecification»
«Interface»
Interface_StereoMatching_Right
stereo_matching(1:img_left)
«clientServerSpecification»
«Interface»
Interface_StereoMatching_Left
stereo_matching(2:img_right)

Figure 17 Inheritance between interfaces

3.4 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 18).

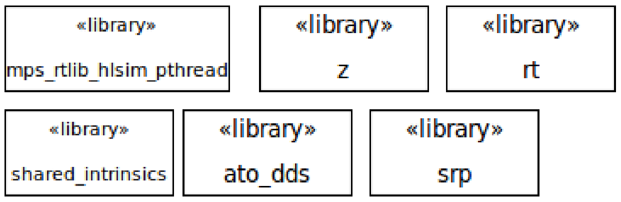


Figure 18 Libraries

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

3.5 Auxiliary Files

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

The *FilesFolder* stereotype has the following attributes:

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.

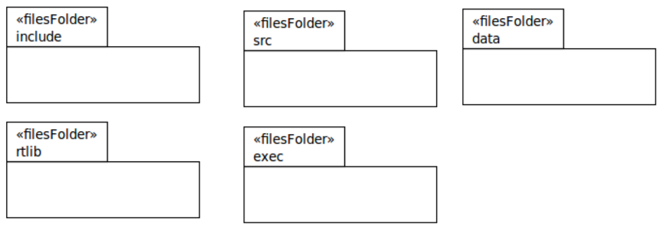


Figure 19 Auxiliary *FilesFolder* packages

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

4 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.

4.1 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 different communicatuion semantics

4.1.1 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.

4.1.2 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 20 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 or it is stored in the channel.

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 synchornized in the function return and to be dealt with as an ordered set.

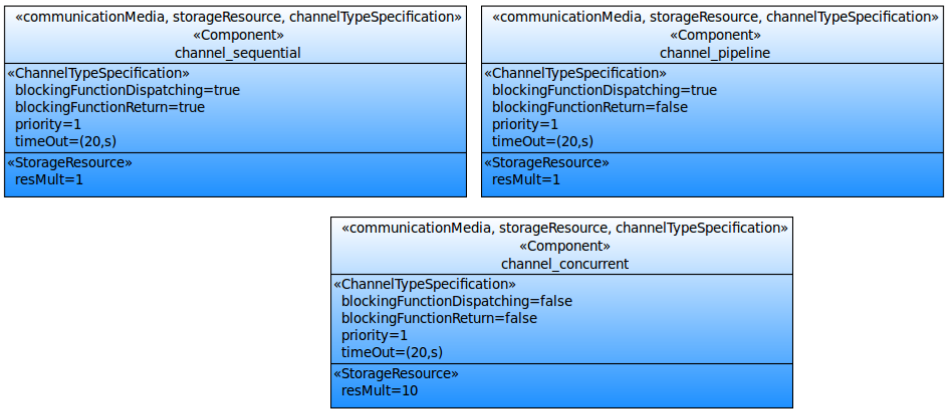

«communicationMedia, storageResource, channelTypeSpecification»
«Component»
channel_sequential
«ChannelTypeSpecification»
blockingFunctionDispatching=true
blockingFunctionReturn=true
priority=1
timeOut=(20,s)
«StorageResource»
resMult=1
«communicationMedia, storageResource, channelTypeSpecification»
«Component»
channel_pipeline
«ChannelTypeSpecification»
blockingFunctionDispatching=true
blockingFunctionReturn=false
priority=1
timeOut=(20,s)
«StorageResource»
resMult=1
«communicationMedia, storageResource, channelTypeSpecification»
«Component»
channel_concurrent
«ChannelTypeSpecification»
blockingFunctionDispatching=false
blockingFunctionReturn=false
priority=1
timeOut=(20,s)
«StorageResource»
resMult=10

Figure 21 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.

Static properties		**Run-Time State**		**Behaviour (Semantics)**			
Of channel		**Of channel**	**OfCalled RtUnit**	**channel**	**Caller (Client) RtUnit**		**Called (Server) RtUnit**
Blocking Function Dispatching	**Blocking Function Return**	**Room for a Call**	**Schedule Resource Available**	**Call Stored**	**Block on Call**	**Block on Return**	**Executed**
true	true	Yes	Yes	No	No	Yes	Yes
true	true	Yes	No	Yes	No	Yes	Delayed
false	true	Yes	Yes	No	No	Yes	Yes
false	true	Yes	No	Yes	No	Yes	Delayed
false	false	Yes	Yes	No	No	No	Yes
false	false	Yes	No	Yes	No	No	Delayed
true	false	Yes	Yes	No	No	No	Yes
true	false	Yes	No	Yes	No	No	Delayed
true	true	No	Yes	No	No	Yes	Yes
true	true	No	No	No	Yes	Yes	Delayed
false	true	No	Yes	No	No	Yes	Yes
false	true	No	No	No	No	No (*)	No
false	false	No	Yes	No	No	No	Yes
false	false	No	No	No	No	No	No
true	false	No	Yes	No	No	No	Yes
true	false	No	No	No	Yes	No	Delayed

Table 3 Communication semantics to be implemented

4.2 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.

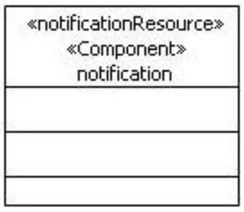


Figure 22 Notification resource

4.3 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*. This property (Figure 23, attribute "type") should de typed by a DataType (Figure 23, Float) included in the *DataView*. Then, in the stereotype attribute *identifierElements* this property should be attached.

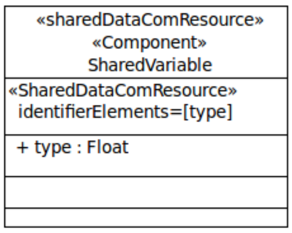


Figure 23 Shared variable

5 Application View

This view serves to capture the application. A component-based approach is used to capture the application model. The application model is captured as a component, which in turn can be composed of other components. Three types of components are supported, active, passive and composite components. An application component communicates with other application components through client-server ports. These ports have associated required/provided interfaces. Provided interfaces declare the functionalities implemented by the component and accessable by other components. Required interfaces declare the functionalities invoked by the component but implemented by others. The application view serves to declare and define these components and to interconnect them, eventually generating the “top” application component, called system component in the application view context. The system component (and by extension, a composite component) is described through the instantiation and interconnection of declared application components. All these instances and interconnections configure the application architecture. Application components are interconnected through channels. A channel can be captured as a simple port-to-port connector (an implicit semantic is assigned). Channel semantics can be configured (the connector is decorated with a stereotyped whose attributes enable such a configuration).

Source code can be associated to the Application View. This is done by associating specific functions and the files containing them the application components. Additionally, paths to those files to complete the link can be provided. In any case, the application model shall be platform independent.

To sum up, this view includes:

- Application architecture and declaration of the application components instanced on it.
- association of source code (as *Files* previously captured in the *FunctionalView*) to the application components (allowing the specification of paths)
- association of libraries

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. They can be stereotyped with <<Channel>> for configuring the specific semantics of the channel.
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.

5.1 Active Components

Active application components are modelled as UML components with the MARTE stereotype <<RtUnit>> (Figure 24). In short, this type of components will be named *RtUnit* component or *RtUnit*. A *RtUnit* component has its own execution threads, its associated C files, and will provide/require services to/from other application components by means of provided and required interfaces. These provided/required interfaces and C files are defined in the *FunctionalView*. A *RtUnit* component can have an associated set of threads in order to execute some specific functions concurrently.

5.1.1 Application Component Attributes

The following attributes of the <<RtUnit>> stereotype are considered (Figure 24):

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

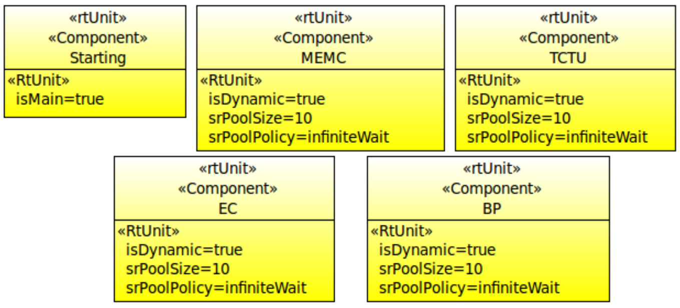


Figure 24 Application components.

5.1.2 Main function of the Application Component

In order to define the main function of the application, the "main" attribute of the <<RtUnit>> stereotype is a used. The attribute is assigned a UML operation captured in the functional view (Figure 25).

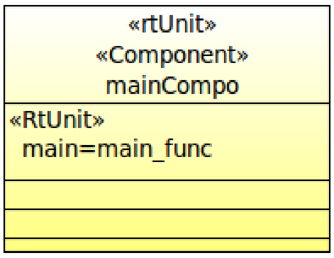


Figure 25 Main function of an application component.

The fact of defining the *main* function of the component involves that the component has implicitly associated a static thread (Figure 25) that executes such a function.

In this case, the main function can have associated specific, initial values to its parameters. In order to annotate thoso values, a UML constraint is used. The constraint has to be owned by the *System* component of the view. In the constraint, the name of the functions and the values of their parameters is captured by means of the following syntax: "$initValue=nameFunction(value1,value2,value3)".

5.1.3 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 26).

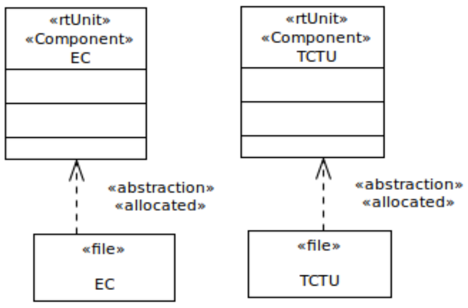


Figure 26 Association Files-Application components

5.1.4 *Association of File Folders with Components*

The application components 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>>.

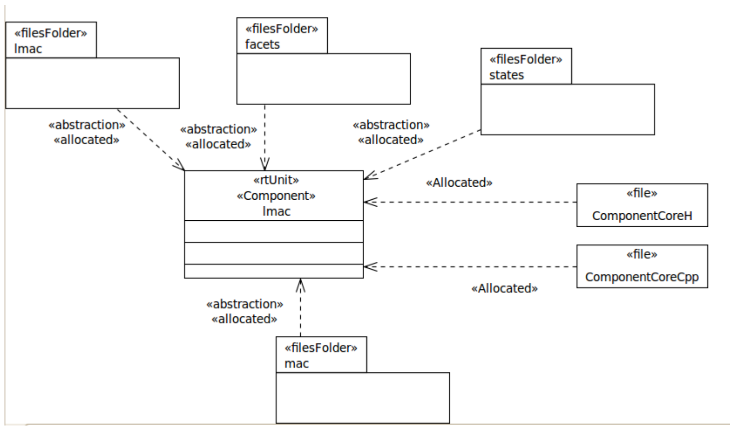
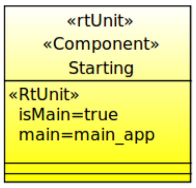
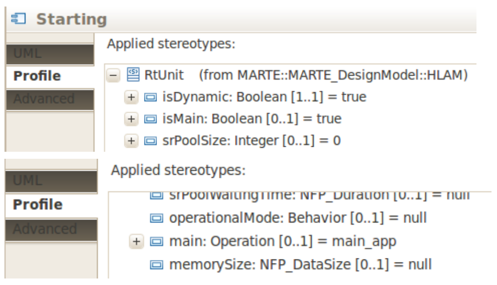


Figure 27 Associations of FileFolders with an Application Component

5.1.5 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*.





Applied stereotypes:
RtUnit (from MARTE::MARTE_DesignModel::HLAM)
isDynamic: Boolean [1..1] = true
isMain: Boolean [0..1] = true
srPoolSize: Integer [0..1] = 0

Applied stereotypes:
srPoolWaitingTime: NFP_Duration [0..1] = null
operationalMode: Behavior [0..1] = null
main: Operation [0..1] = main_app
memorySize: NFP_DataSize [0..1] = null

Figure 28 Main application component

5.1.6 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.

5.1.7 Connectors

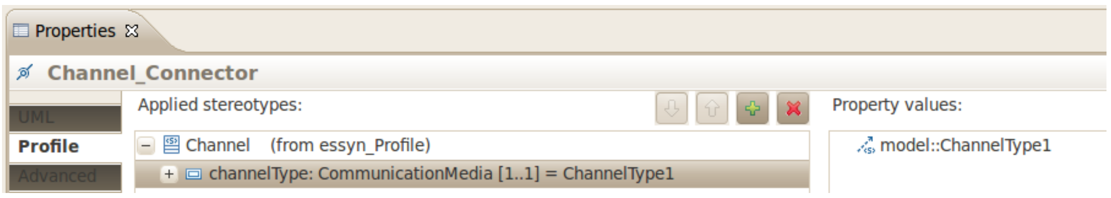
The ports are connected by using UML connectors. The conectors can represent simple connections or communication channels.

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*.

This sterotype has the attribute *channelType* (Figure 29) that is typed by a CommunicatioMedia component for representing tthe channel type and which is defined in the *CommunicationView*.



Channel_Connector

Applied stereotypes:

Channel (from essyn_Profile)

channelType: CommunicationMedia [1..1] = ChannelType1

Property values:

model::ChannelType1

Figure 29 Channel type attached to the *Channel* connector

Only UML assembly connectors (in Figure 30 the UML connector established between the elements “imageAcquisition” and “imagePreProcessing”) should be stereotyped by the *Channel*. The *UML delegation* connectors (in Figure 30 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.

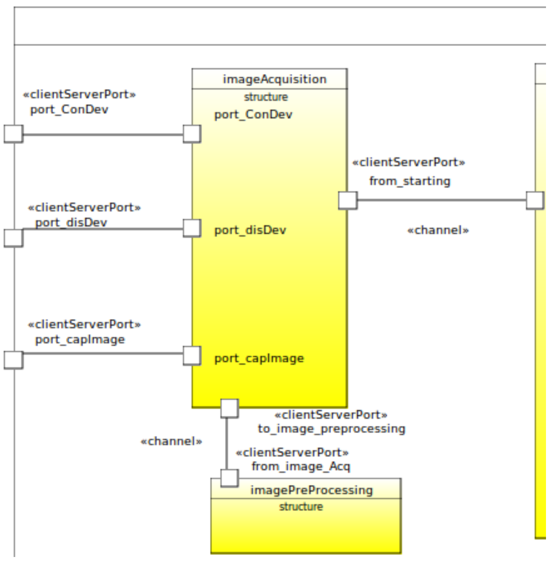


Figure 30 Assambly and delegation connectors

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 is 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 31).

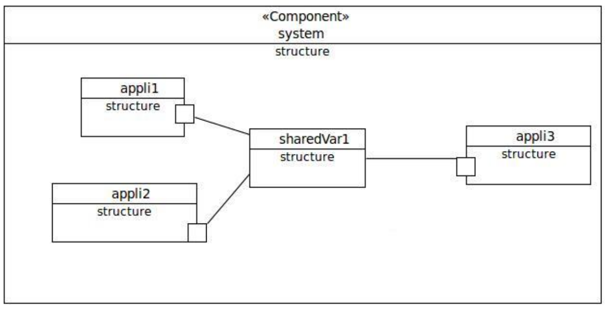


Figure 31 shared variable used by several application components

5.2 Pasive Components

A different set of component can be captured: components that represent information shared by several components and whose concurrent access must be protected by some synchronization mechanism. These components are stereotyped with the MARTE <<*PpUnit*>>. It is a passive element.

The access semantics associated to the *PpUnit* component is defined by the attribute *concPolicy***.**This attribute establishes the policy applied to the services provided by *PpUnit*. It may take the following values:

- *sequential*: no concurrency mechanism is associated so the system designer must assure that no concurrent invocations are produced.
- *guarded*: several invocations may occur concurrently, but only one is attended at a time. The rest are blocked until the execution of the invocation being attended finishes.
- *concurrent*: several invocations may occur and be attended at the same time.

The services provided by the *PpUnit* are enclosed in interfaces and offered by provided *ClientServerPort*s. All the interfaces provided by a *PpUnit* component inherent the value of the attribute *concPolicy* of such *PpUnit* component.

As in the case of the *RtUnit* components, they can have associated files, files folder, libraries, and the definition of paths language…

There is a modelling constraint: the channels connected to the PpUnit instances must blocking since the PpUnit is a passive element and it does not have resources for attending income calls but the resource of the calling element.

5.3 Composite Components

The methodology enables to model composite application components. These components have an internal structure, composed of interconnected application components. Composite components are specified as UML components decorated with the UML standard stereotype <<Subsystem>>.

The internal structure is captured through a composite structure diagram associated to the *Subsystem* component. In this diagram, instances of application components or other composite component instances are created and connected via port-to-port connections.

The *Subsystem* components has ports. These ports are connected to the ports of the internal application instances. The name of a ports of the *Subsystem* components (parent ports) should have the same as the name of the port of the internal application component instance it is connected to (children port). Parent and children ports shall have the same interface and the same type of interface (both required or both provided).

The connectors between a parent port and a children port must not be stereotyped (assembly connectors). Only connectors connecting internals application components can be specified as channels (applying to them the <<Channel>> stereotype).

Subsystem components are not expected to have any *File*, *FileFolder, or library* library associated.

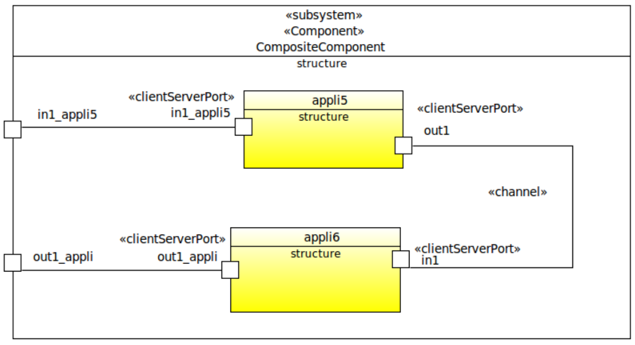


Figure 32 Composite Component

5.4 Application Architecture

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

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

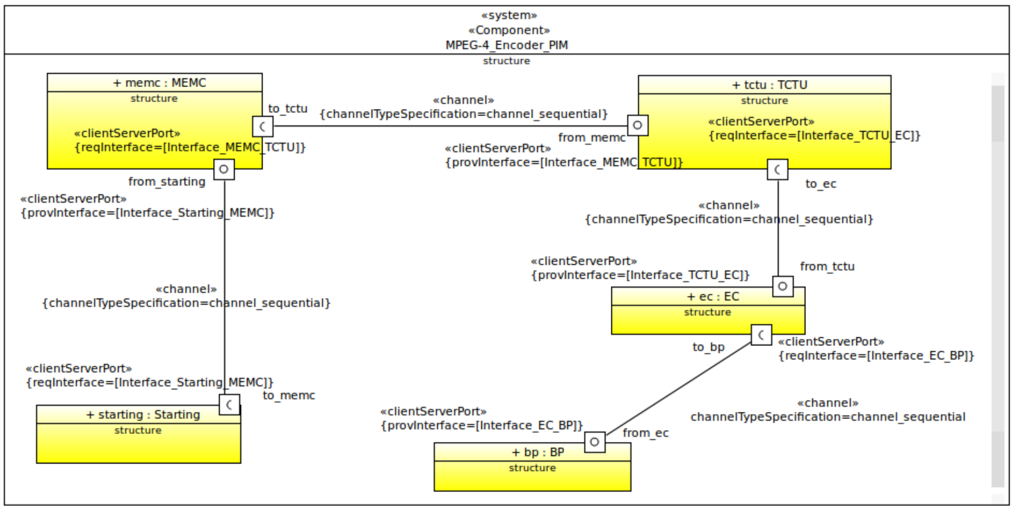

«system»
«Component»
MPEG-4_Encoder_PIM
structure
+ memc : MEMC
structure
«clientServerPort»
{reqInterface=[Interface_MEMC_TCTU]}
to_tctu
«channel»
{channelTypeSpecification=channel_sequential}
from_memc
«clientServerPort»
{provInterface=[Interface_MEMC_TCTU]}
+ tctu : TCTU
structure
«clientServerPort»
{reqInterface=[Interface_TCTU_EC]}
to_ec
from_starting
«clientServerPort»
{provInterface=[Interface_Starting_MEMC]}
«channel»
{channelTypeSpecification=channel_sequential}
«clientServerPort»
{provInterface=[Interface_TCTU_EC]}
from_tctu
+ ec : EC
structure
«channel»
{channelTypeSpecification=channel_sequential}
to_bp
«clientServerPort»
{reqInterface=[Interface_EC_BP]}
«clientServerPort»
{reqInterface=[Interface_Starting_MEMC]}
to_memc
+ starting : Starting
structure
«clientServerPort»
{provInterface=[Interface_EC_BP]}
«channel»
channelTypeSpecification=channel_sequential
from_ec
+ bp : BP
structure

Figure 33 Application Structure 1

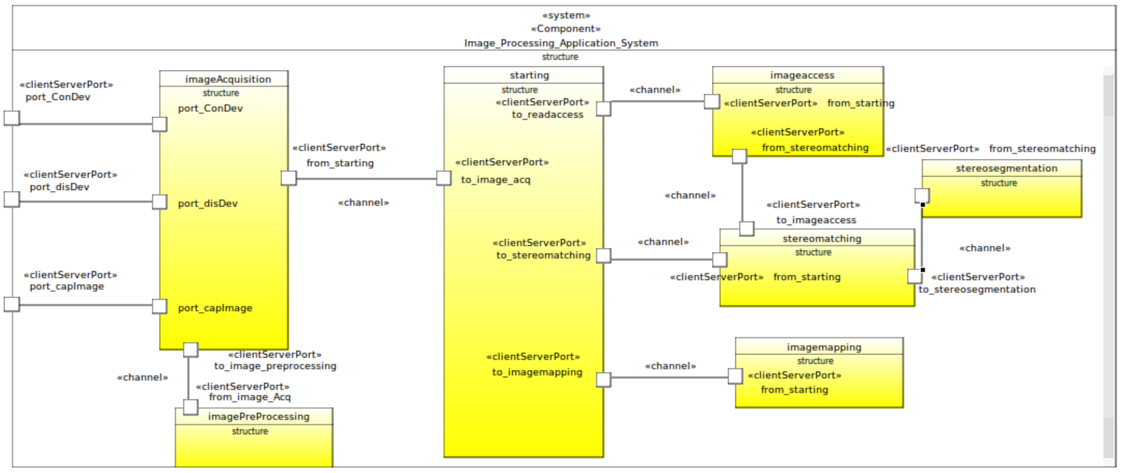

«system»
«Component»
Image_Processing_Application_System
structure
«clientServerPort» port_ConDev
«clientServerPort» port_disDev
«clientServerPort» port_capImage
imageAcquisition
structure
port_ConDev
port_disDev
port_capImage
«clientServerPort» from_starting
«channel»
«clientServerPort» to_image_preprocessing
«channel»
«clientServerPort» from_image_Acq
imagePreProcessing
structure
starting
structure
«clientServerPort» to_readaccess
«clientServerPort» to_image_acq
«clientServerPort» to_stereomatching
«clientServerPort» to_imagemapping
«channel»
imageaccess
structure
«clientServerPort» from_starting
«clientServerPort» from_stereomatching
«clientServerPort» from_stereomatching
stereosegmentation
structure
«channel»
«clientServerPort» to_imageaccess
«channel»
stereomatching
structure
«clientServerPort» from_starting
«channel»
«clientServerPort» to_stereosegmentation
«channel»
imagemapping
structure
«clientServerPort» from_starting

Figure 34 Application Structure 2

5.4.1 *System ports: I/O communication*

The *System* component communicates with the external environment. This environment communication is established through ports. These UML ports should be specified by the MARTE stereotype <<ClientServerPort>> (Figure 30 and Figure 34), specifying the correct values of the attribute *kind*, *provInterface* and *reqInterface* in the case the communication is dealt with using function calls. in Others (shared variable and synchronization mechanism), the ports are not stereotyped.

These *System* ports are connected to application instances. This connection is port-to-port. In order to keep consistence, the system ports connected to application instance ports should have the same name (Figure 30 and Figure 34). The connection between the *System* port and the application port is never stereotyped (Figure 30 and Figure 34) since it does not represent a real channel, so the stereotype <<Channel>> must no be applied on these connectors.

5.4.2 Periodic Application Instances

An application instance can be characterized by a period, triggering its execution according to that period.

The period of an application component is modelled by a UML comment specified by the MARTE stereotype <<RtSpecification>>. In the attribute *occKind* the period is annotated as:

- periodic (period= (value, unitTime))

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

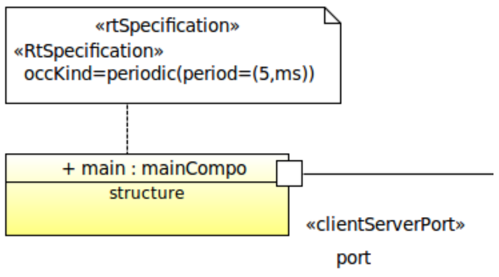


Figure 35 Periodic application instance

5.4.3 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 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 36.

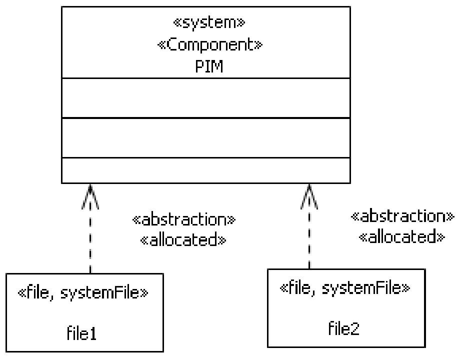


Figure 36 *System* component with files associated

5.5 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 37 shows.

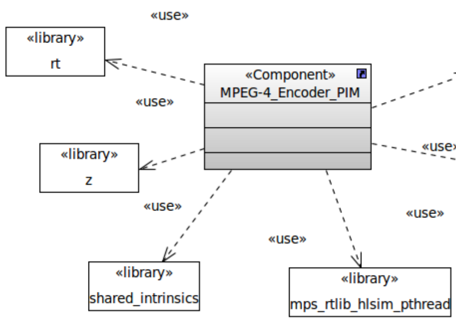

«use»
«library» rt
«Component» MPEG-4_Encoder_PIM
«library» ato_dds
«use»
«library» z
«use»
«library» srp
«use»
«use»
«use»
«library» shared_intrinsics
«library» mps_rtlib_hlsim_pthread

Figure 37 ***System*** **component with libraries associated**

5.6 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.

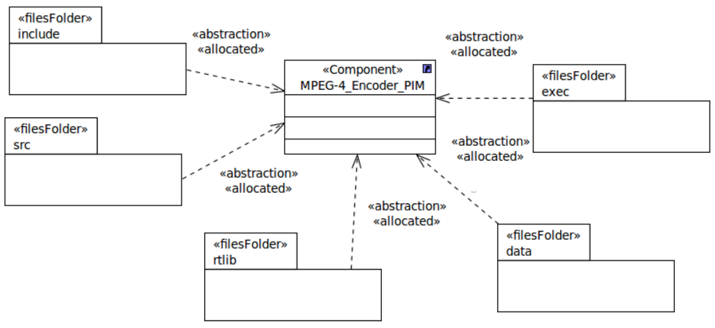

«filesFolder» include
«abstraction» «allocated»
«abstraction» «allocated»
«Component» MPEG-4_Encoder_PIM
«filesFolder» exec
«filesFolder» src
«abstraction» «allocated»
«abstraction» «allocated»
«abstraction» «allocated»
«filesFolder» rtlib
«filesFolder» data

Figure 38 ***System*** **component with** ***FileFolder*** **package**

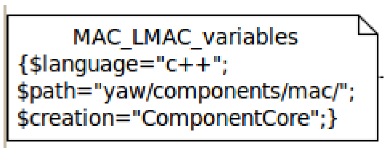
5.7 Modelling Variables

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++".

5.8 Modeling Variable Specification

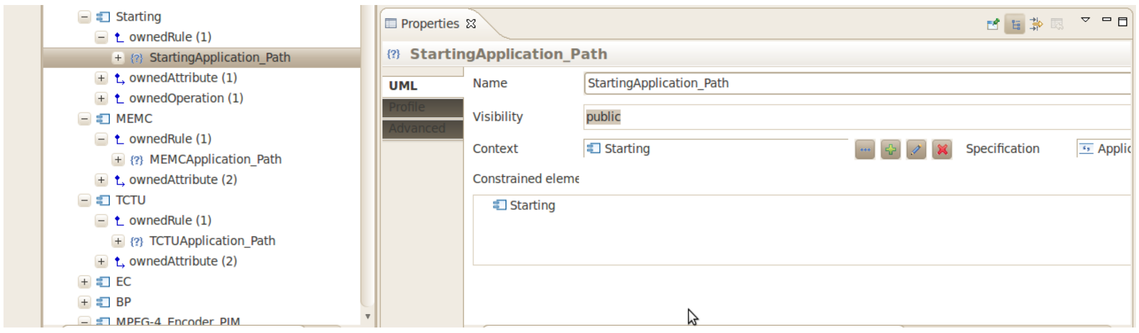
The variables are annotated as *$nameVariable=”valueVariable”;* as Figure 39 shows.



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

Figure 39 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 40) and in the “Context” attribute of the constraint (Figure 40).



Starting
ownedRule (1)
StartingApplication_Path
ownedAttribute (1)
ownedOperation (1)
MEMC
ownedRule (1)
MEMCApplication_Path
ownedAttribute (2)
TCTU
ownedRule (1)
TCTUApplication_Path
ownedAttribute (2)
EC
BP
MPEG-4_Encoder_PIM

Properties

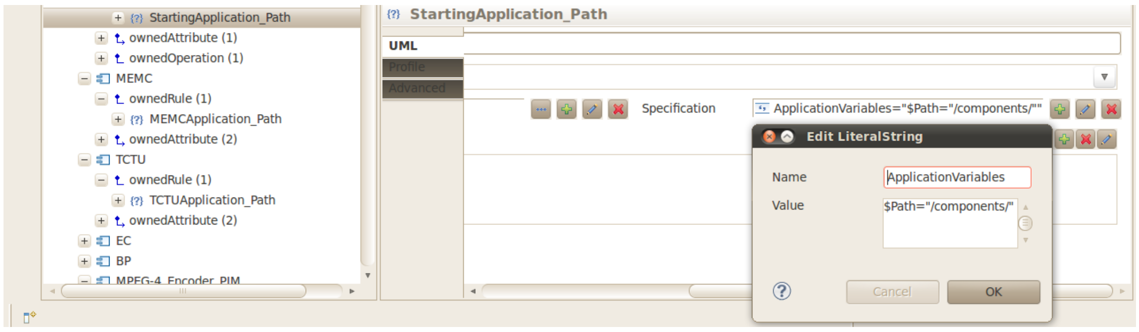
StartingApplication_Path

UML | Profile | Advanced

Name: StartingApplication_Path
Visibility: public
Context: Starting
Specification: Applic
Constrained eleme: Starting

Figure 40 UML constraint for application component variables

The “Specification” attribute of constraint contains the declaration of the variables. The variable annotation is captured in a LiteralString (Figure 41).



StartingApplication_Path
ownedAttribute (1)
ownedOperation (1)
MEMC
ownedRule (1)
MEMCApplication_Path
ownedAttribute (2)
TCTU
ownedRule (1)
TCTUApplication_Path
ownedAttribute (2)
EC
BP
MPEG-4_Encoder_PIM

StartingApplication_Path

UML | Profile | Advanced

Specification: ApplicationVariables="$Path="/components/""

Edit LiteralString

Name: ApplicationVariables
Value: $Path="/components/"

Cancel | OK

Figure 41 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 40). The ConstrainedElement attribute denotes the model element which the variables annotated in the constraint are applied. This association is captured by using and UML link between the constraint and the model element.

It is necessary to distinguish which element is the owner of the constraint and the element to be specified by the variables of the constraint. In Figure 42, there are four constraints (“MAC_LMAC_states_facets”, “MAC_LMAC_varibles”, “MAC_InterfacesFolder_LMAC_common” and “MAC_Folder_LMAC”).

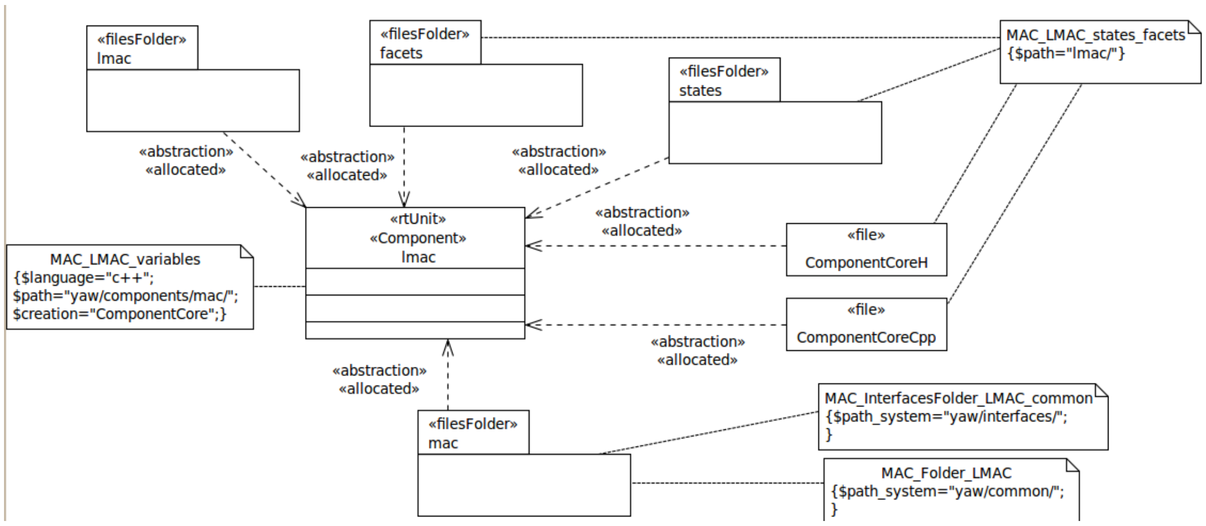


Figure 42 Example of multiple constaints in the same application component

All these UML constraints are owned by the application component "lmac" (Figure 43).

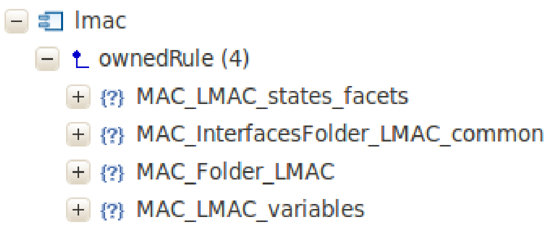
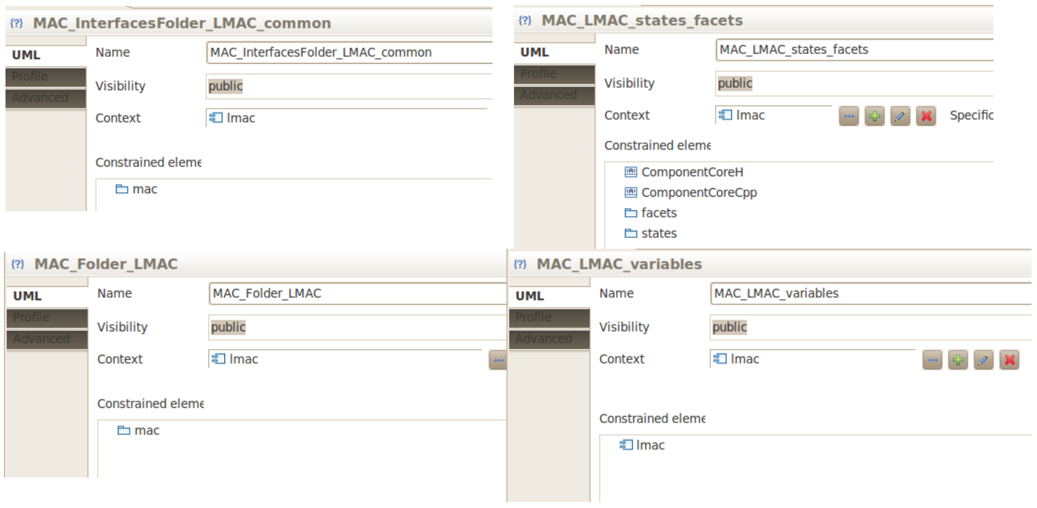


Figure 43 Constrains of the "lmac" application component

However, not all of these constraints are applied to the same model element, denoted by the attribute "ConstrainedElement" of the constraints (Figure 44).



MAC_InterfacesFolder_LMAC_common
UML
Profile
Advanced
Name MAC_InterfacesFolder_LMAC_common
Visibility public
Context lmac
Constrained eleme
mac

MAC_LMAC_states_facets
UML
Profile
Advanced
Name MAC_LMAC_states_facets
Visibility public
Context lmac
Specific
Constrained eleme
ComponentCoreH
ComponentCoreCpp
facets
states

MAC_Folder_LMAC
UML
Profile
Advanced
Name MAC_Folder_LMAC
Visibility public
Context lmac
Constrained eleme
mac

MAC_LMAC_variables
UML
Profile
Advanced
Name MAC_LMAC_variables
Visibility public
Context lmac
Constrained eleme
lmac

Figure 44 Constraints with different constrained elements

5.8.1 System Components

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

1. language

2. path

5.8.2 Language

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

5.8.3 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 45), which act as base paths for the rest of the system. This is mandatory.

5.9 Association of source code to 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

5.10 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.

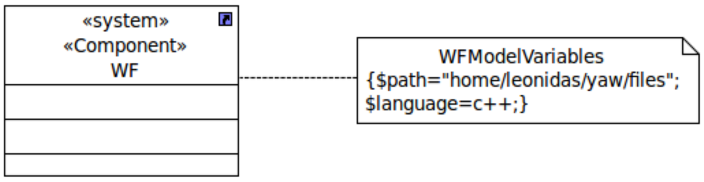


Figure 45 Specification of the System's base path

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 45).

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

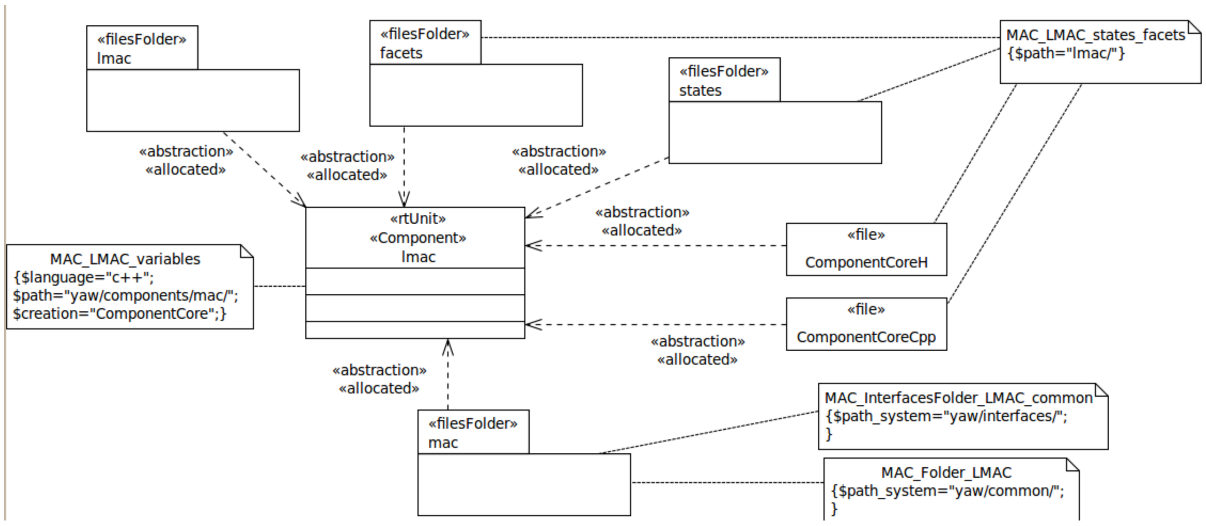

«filesFolder» lmac
«filesFolder» facets
«filesFolder» states
MAC_LMAC_states_facets {$path="lmac/"}
«abstraction» «allocated»
«rtUnit» «Component» lmac
MAC_LMAC_variables {$language="c++"; $path="yaw/components/mac/"; $creation="ComponentCore";}
«file» ComponentCoreH
«file» ComponentCoreCpp
«filesFolder» mac
MAC_InterfacesFolder_LMAC_common {$path_system="yaw/interfaces/"; }
MAC_Folder_LMAC {$path_system="yaw/common/"; }

Figure 46 Application components with different types of model variables

To complete the path of the files “ComponentCoreH” and “ComponentCoreCpp” in Figure 46, to the previous path (“home/leonidas/yaw/files/components/mac/”), the path associated with the Files is concatenated as well: “home/leonidas/yaw/files/components/mac/lmac/”. Finally, the name of the attribute “File name” of the *File* model element (see section 3.1) is 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 45) plus the application component path (Figure 46) 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 *$path_system* variable is annotated. In this, the creation of the path does not consider the base path of the application component (in Figure 46, “yaw/components/files/”). In this case, the *System* path (Figure 45) is concatenated with the value of the *$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 46, “mac” *FilesFolder*).

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 47).

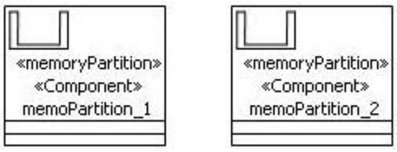


Figure 47 Memory partitions

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 48).

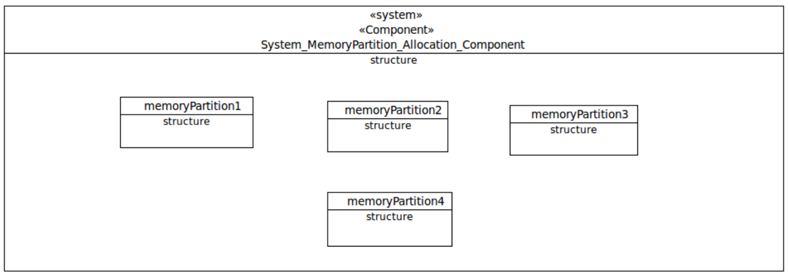


Figure 48 Executables definition

6.3 Application Allocation structure

In this view, the allocation 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 49).

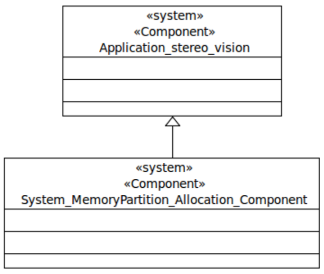

«system»
«Component»
Application_stereo_vision
«system»
«Component»
System_MemoryPartition_Allocation_Component

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

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

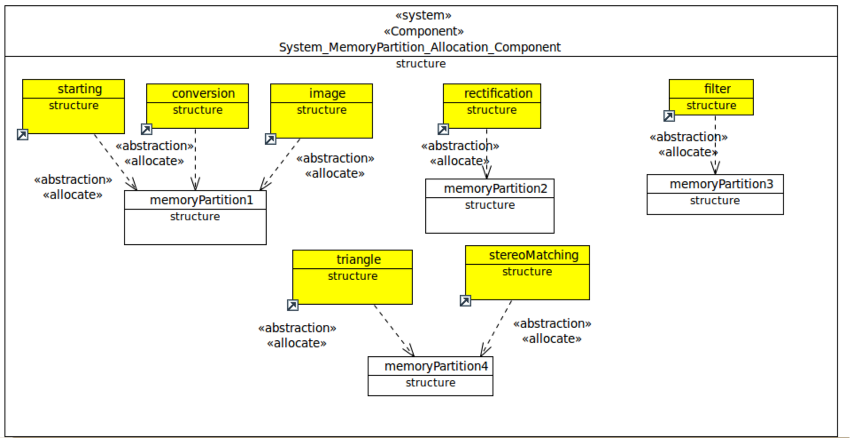


Figure 50 Memory partition allocation

In Figure 50, the yellow boxes are application components that are mapped onto memory partitions.

6.3.1 *Contraints of Allocation*

There is a modelling constraint in terns of application mapping: for each memory space, only one-application instance with implicit thread associated (section 5.1.2) is allowed.

6.4 Composite components Allocation

When an instance of a composite component is allocated in a memory partition, it is involved that all the internal instances of such composite component are allocated in that memory partition. The internal parts of a composite component can not be allocated in different memory partitions.

PDM Views

7 HW Resources View

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

The UML elements used in this view are:

1. 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 HwBus HwMedia HwEndPoint HwBridge HwI_O HwPLD HwISA

Table 4 MARTE stereotypes used for refining the HW platform

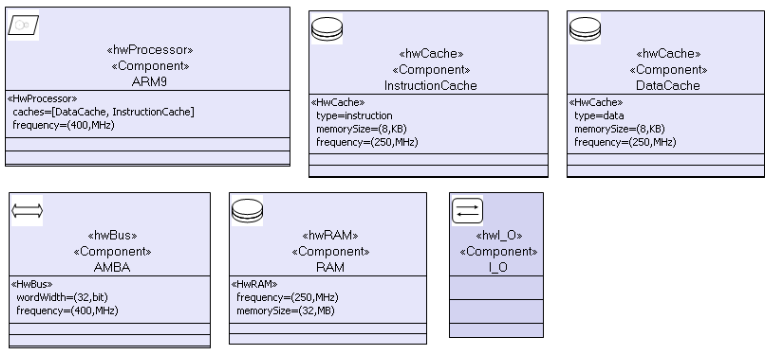


Figure 51 HW platform resources

7.1 Physical Magnitudes

HW component attributes can be annotated with values, which can be either a-dimensional or represent a physical magnitude. There are two different ways to annotate the value of the attribute with its corresponding physical magnitude:

1. (**value=***valueSpecification***, units=***physicalMagnitude***)**
   a. (**value**=200, **units**=MHz)
   b. (**value**=2, **units**=mW)
2. *(valueSpecification, physicalMagnitude)*
   a. (200,GB)
   b. (25,nJ)

The accepted units for the each attribute and the default physical magnitude are shown in the following table.

Attribute	Physical magnitude
frequency	GHz MHz KHz Hz
memorySize	TB GB MB KB B
wordWidth	byte
BandWidth	Gb/s Mb/s Kb/s b/s
memoryLatency	us Ns
power	W mW uW nW

	pW
energy	J mJ uJ nJ pJ
blockSize	word

Table 5 HW attributes and physical units

7.2 HW Processors

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

Frequency

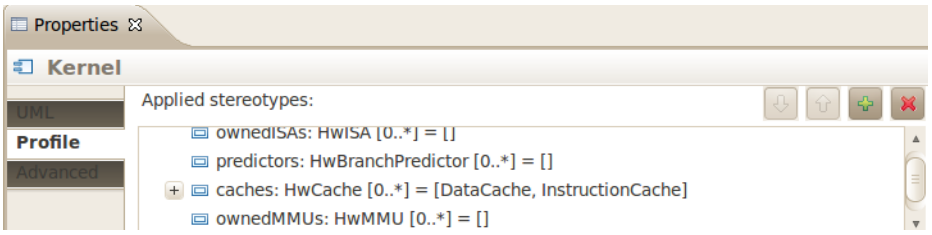
The frequency of the processors is captured in the *HwProcessor* attribute *frequency*.

Slots

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

7.2.1 Cache Processor

Each HW processor could have data and instruction caches memories. Thus, each HW processor can have associated a set of *HwCaches* instances. The caches can be associated to a *HwProcessor* by means of the attribute *caches* of the stereotype *HwProcessor* (Figure 52). This stereotype attribute selects the UML components that are characterized by *HwCaches*.



ownedISAs: HwISA [0..*] = []
predictors: HwBranchPredictor [0..*] = []
caches: HwCache [0..*] = [DataCache, InstructionCache]
ownedMMUs: HwMMU [0..*] = []

Figure 52 Associating caches to a HWProcessor

7.2.2 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 considered in this 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, CortexA9, undef.

7.3 Processor Caches

The cache memories are modelled by the MARTE stereotype *HwCache*. Table 6 shows the possible values of the *type* and *level* attribute of the *HwCache* stereotype that determines the type of cache. Figure 53 shows an example of caches components.

HwCache attribute	Type of Cache

level = 1
&
type = data
	Data cache

level = 1
&
type = instruction
	Instruction Cache

level !=1 & type = unified	Unified Cache for caches of level more that one

Table 6 HwCache attribute values

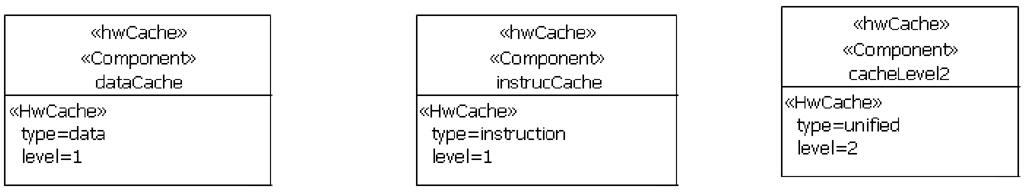


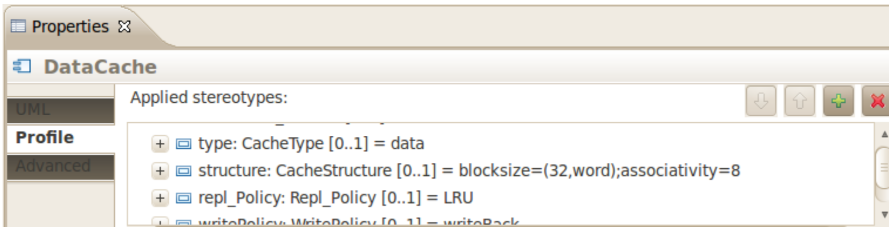
Figure 53 Cache components

Additionally, the caches can be characterized with three additional attributes: the block size (specifies the width of a cache block), the associativity and the number of sets. These caches attributes can be specified in the attribute *structure* of the MARTE stereotype *HwCache*. The attribute *structure* is typed as *CacheStructure* (Table 7).

***HwCache* attribute**	**Attributes**
structure	blockSize associativity

Table 7 Definition of the *structure* attribute

The specification of these attributes has to be annotated as a string. The attributes annotation is shown in Figure 54. The attributes are identified as *blockSize* and *associtivity*. Both data annotations are specified in parentheses and separated by comma. The unit of the *blockSize* is the WORD (Figure 54). The word size associated to the cache memory is annotated in a UML property named *wordSize* of the *HwCache* component. This specifed by the MARTE stereotype <<Nfp>> and typed by the MARTE NFP data type NFP_DataSize (Figure 55). Then, in the "Default Value" the value is annotated. When this attribute is not present, the default value annotated is 4 Bytes.



type: CacheType [0..1] = data
structure: CacheStructure [0..1] = blocksize=(32,word);associativity=8
repl_Policy: Repl_Policy [0..1] = LRU

Figure 54 Specification of the attributtes *blockSize* and *associativity*

The size of the caches is defined in the attribute *memorySize*.

The type of write policy is specified in the attribute *writePolicy*. It can be *writeBack* or *writeThrough*.

In the case the cache is typed as instruction (attribute type), another attribute can be captured; the size of the address. This property is annotated in the *HwCache* attribute *addressSize*.

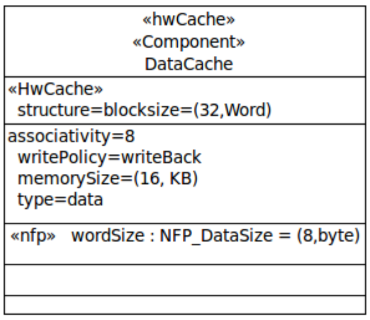


Figure 55 Cache specification

7.4 Bus

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

Word width

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

Band width

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

Burts size

The property burst size denotes the size … It is modelled by adding an UML property to the *HwBus* component named it “burstSize”. The attribute is specified by the MARTE stereotype <<NFP>>. This attribute is typed by the MARTE NFP data type NFP_DataSize. Then, in the “Default Value” property the value is annotated. When this attribute is not present, the default value annotated is the *wordWidth* attribute value.

7.4.1 TDMA bus

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

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

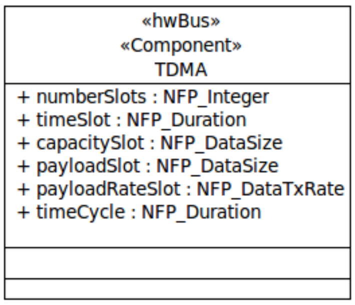

«hwBus»
«Component»
TDMA
+ numberSlots : NFP_Integer
+ timeSlot : NFP_Duration
+ capacitySlot : NFP_DataSize
+ payloadSlot : NFP_DataSize
+ payloadRateSlot : NFP_DataTxRate
+ timeCycle : NFP_Duration

Figure 56 TDM bus component properties

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

7.5 Bridges

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

7.6 FPGA

The FPGA is modeled by the MARTE stereotype <<HwPLD>>.

7.7 Memories

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

Memory size

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

Memory latency

The memory latency attribute is annotated in the attribute *timmings*. In this attribute, there is annotated *memoryLatency*=(value, unit).

7.8 Network

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

7.9 Network Interfaces

The network interfaces are modelled by the MARTE stereotype <<HwEndPoint>>. 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.

7.10 I/O Components

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

7.11 HW components' Functional Modes

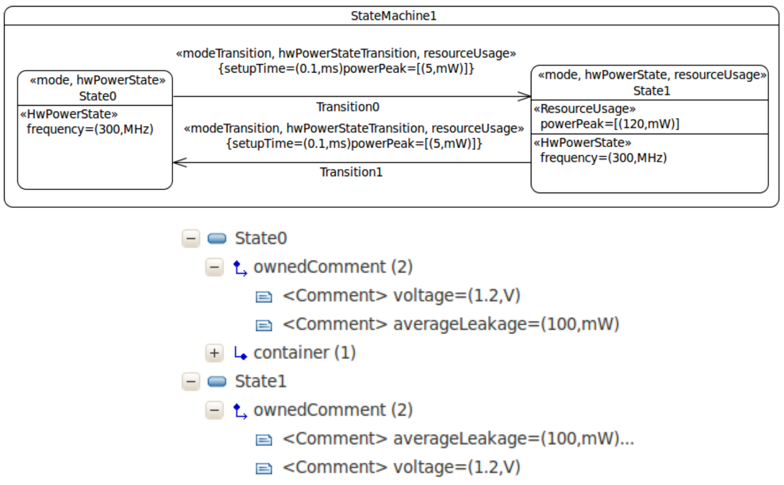
The HW components can have different associated functional modes that specify different characteristics that define the HW component's behaviour according to a set of configuration parameters. These functional modes are defined by attributes: *frequency*, *voltage*, *dynamic power* and *average leakage*. 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 attribute *Pstatic* of the HwPowerState enables to capture the power consumption in idle in this mode. The dynamic power of the mode is defined by the application of the MARTE stereotype <<ResourceUsage>>, specifying the attribute *powerPeak*. In order to define the last two attributes of a functional mode, *voltage* and *average leakage*, two UML comments should be associated with the corresponding UML state. There, both values are annotated. All the attribute values should be annotated as the MARTE specifies in order to define the non-functional properties (value, unit).

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

[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.


StateMachine1
«modeTransition, hwPowerStateTransition, resourceUsage»
{setupTime=(0.1,ms)powerPeak=[(5,mW)]}
«mode, hwPowerState»
State0
«HwPowerState»
frequency=(300,MHz)
Transition0
«modeTransition, hwPowerStateTransition, resourceUsage»
{setupTime=(0.1,ms)powerPeak=[(5,mW)]}
Transition1
«mode, hwPowerState, resourceUsage»
State1
«ResourceUsage»
powerPeak=[(120,mW)]
«HwPowerState»
frequency=(300,MHz)

```
State0
  ownedComment (2)
    <Comment> voltage=(1.2,V)
    <Comment> averageLeakage=(100,mW)
  container (1)
State1
  ownedComment (2)
    <Comment> averageLeakage=(100,mW)...
    <Comment> voltage=(1.2,V)
```

Figure 57 HwProcessor mode specification

7.12 Power Consumption

The HW components have associated static power consumption. This value is modelled by applying the MARTE stereotype <<HwComponent>> to the HW component and annotating the value in the attribute *staticConsumption*.

7.13 Energy Consumption

There are set of properties that can be associated to specific HW resources in order to determine the energy consumption of some actions implemented by these HW resources.

Processors

The processors have associated the energy consumed by cycle. The energy consumption per cycle is captured by adding a UML attribute named *cycle* typed by a NFP_Energy data type. The attribute is specified by the MARTE stereotype <<NFP>>. Then, in the property “Default Value” the value is annotated.

Caches

The caches have associated two energy consumptions; the consumption of a hit and the consumption of a miss. The hit energy consumption is captured by adding a UML attribute named *hit* typed by a NFP_Energy data type. The attribute is specified by the MARTE stereotype <<NFP>>. Then, in the property “Default Value” the value is annotated.

The miss consumption is captured by adding a UML attribute named *miss* repeating afore explained process for specifying the value.

Buses

The buses have associated the energy consumed in order to access to them. The bus access energy consumption is captured by adding a UML attribute named *access* typed by a NFP_Energy data type. The attribute is specified by the MARTE stereotype <<NFP>>. Then, in the property "Default Value" the value is annotated.

Memories

The memories have associated the energy consumed in order to access to them. The memory access energy consumption is captured by adding a UML attribute named *access* typed by a NFP_Energy data type. The attribute is specified by the MARTE stereotype <<NFP>>. Then, in the property "Default Value" the value is annotated.

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]
scheduler: Scheduler[*]
drivers: DeviceBroker [*]
interProcessCommunication: InterProcessCommunicationMechanism [1]

Figure 58 OS stereotype attributes (modificar)

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

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

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

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

The *interProcessCommunication* attribute defines the OS services that automatically create the communication infrastructure in order to communicate processes in the OS. Thus, code will be created ad-hoc depending on which mechanism is specified for each OS instance. Five types of inter process communication mechanism are currently supported for automatic code generation. These types are:

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

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

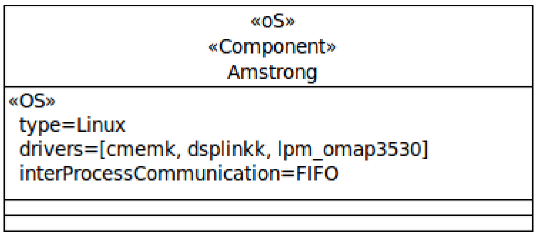

«oS»
«Component»
Amstrong
«OS»
type=Linux
drivers=[cmemk, dsplinkk, lpm_omap3530]
interProcessCommunication=FIFO

Figure 59 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

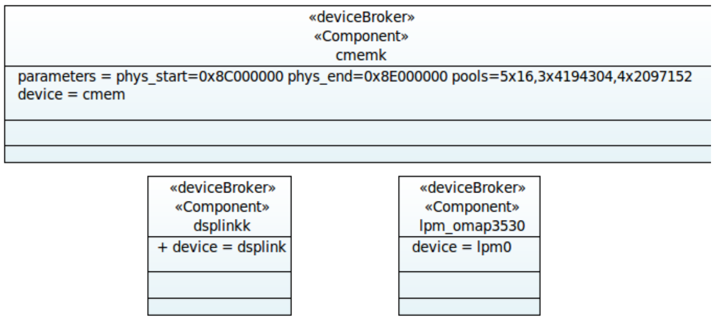


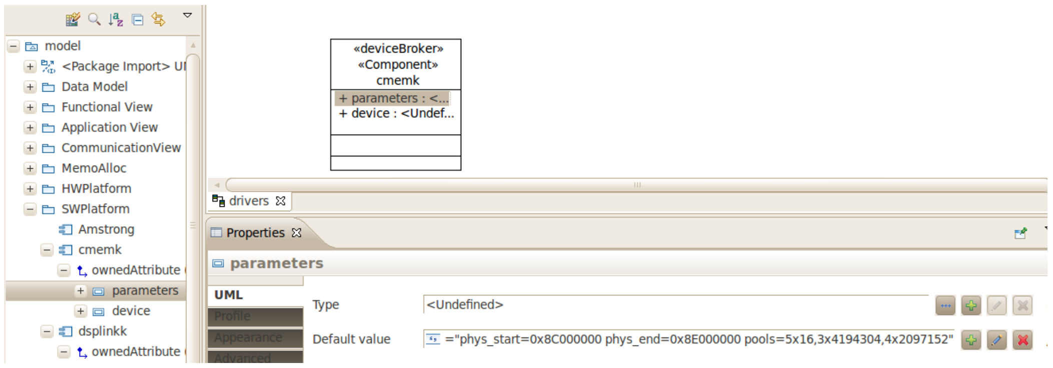
Figure 60 Driver for DSP management

8.1.1 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.

8.1.2 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.



parameters

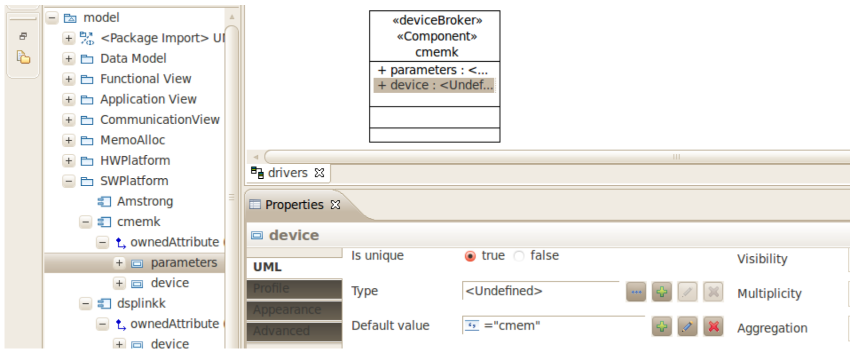
Type: <Undefined>

Default value: ="phys_start=0x8C000000 phys_end=0x8E000000 pools=5x16,3x4194304,4x2097152"

Figure 61 “Parameter” driver property

8.1.3 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.



device

Is unique: [✓ true] false

Type: <Undefined>

Default value: ="cmem"

Figure 62 “Device” driver property.

PSM Views

9 Architectural View

The *Architectural view* captures the platform specific model (PSM) as a mapping of the PIM onto the platform. Moreover, the architectural view also describes the platform architecture. The platform specific model is captured as a component containing the following items:

- SW platform architecture (e.g. OS instances).
- HW platform architecture, which includes
  - Instances of HW resources (processors, memories, buses, network, etc.).
  - Interconnections amon those HW resources
- Association of the HW resources to OS.
- Mapping of the PIM to the platform

9.1 Modelling of the HW/SW platform architecture

The *Architectural View* contains the *System* component, i.e. a component decorated by the <<System>> stereotype. The *System* component of the architectural view represents the platform specific model. Only one *System* component should be present in the *Architectural View* package. A composite structure diagram, as the one shown in Figure 63, is associated the system component, and used to capture the HW/SW architecture of the platform.

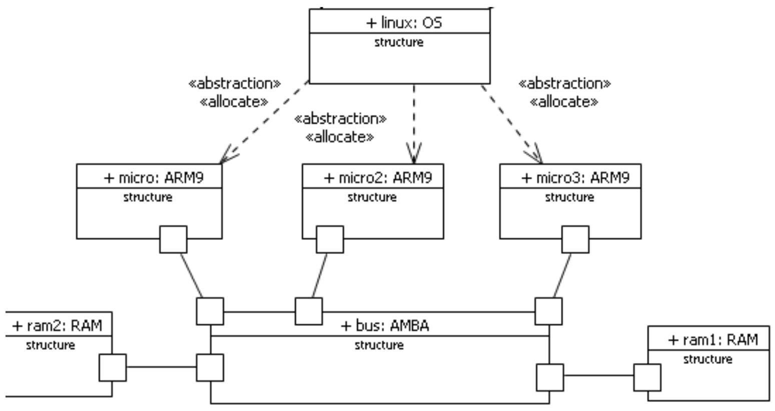


Figure 63 HW & SW platform architectures

The HW architecture is captured by instancing HW components declared in the *HW Resources View* and interconnecting them through UML port-to-port connections. The SW platform architecture is composed of instances of the *OS* components included in the *SWPlatformView*.

9.2 Platform Mapping: SW instances onto HW instances

The association of the *OS* instances with HW resources instances is modelled by means of UML abstractions decorated with the MARTE <<allocate>> stereotype (see Figure 63).

9.3 PIM to Platform Mapping

In the simplest case, application component instances can be mapped to the platform resources (RTOS instances or processing elements). Moreover, the methodology also enables the mapping of other PIM elements, namely schedulable resources (threads) and memory spaces onto the platform resources.

In order to enable the mapping of PIM application component instances onto the SW/HW platform, the *System* component of the architectural view (representing the PSM) view has to be defined as a specialization of a *System* component defined in the PIM view which contains the PIM elements to be mapped. Therefore, if component application instances are going to be mapped, the PIM System component of the *ApplicationView* can be extended. However, if memory spaces are going to be mapped, then the *System* component of the MemorySpaceView have to appear as the parent port. The specialization relation is captured in a UML class diagram, by mean of a UML generalization.

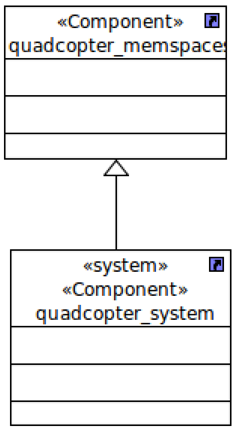


Figure 64 The *System* component of the *Architectural View* reflects a PSM, which specializes and increments the PIM model.

Figure 64 gives an example of inheriting the system component of the memory space view. It allows to refer and map memory partitions (Figure 65). Notice that, since both, the system component of the memory space view, and the system component of the concurrency view (in case they appear) inherit the system component of the application view, in this case, the application components can be also referenced and mapped (Figure 65).

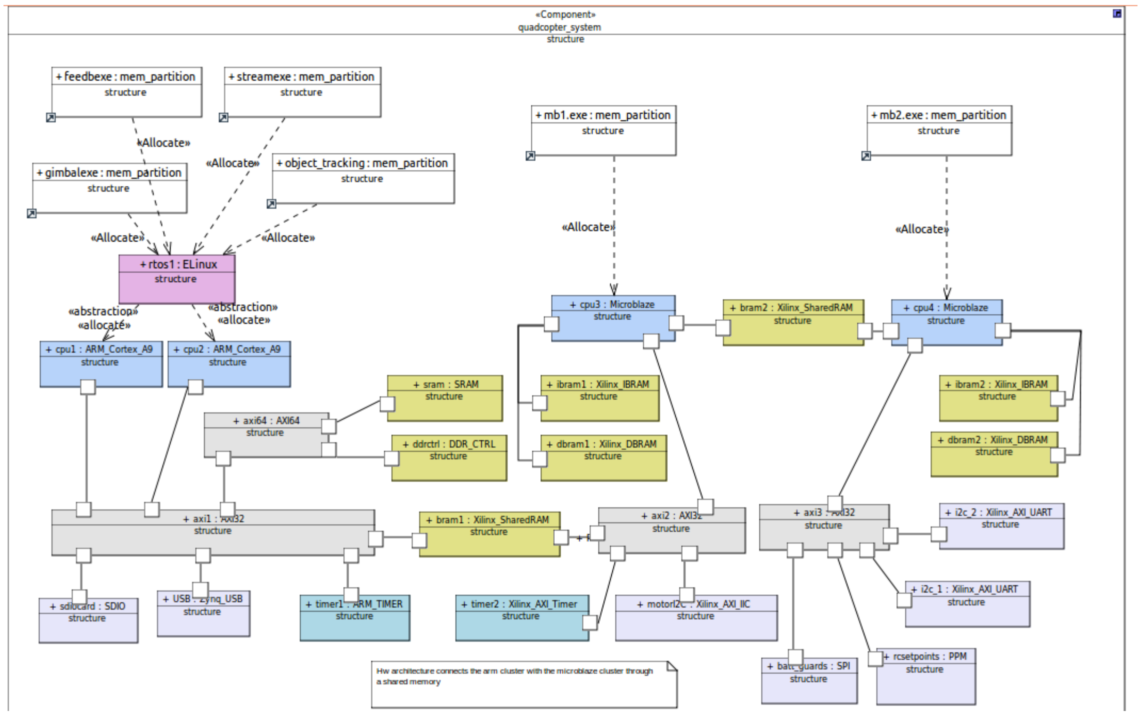


Figure 65 Mapping memory partitions onto the HW/SW platform.

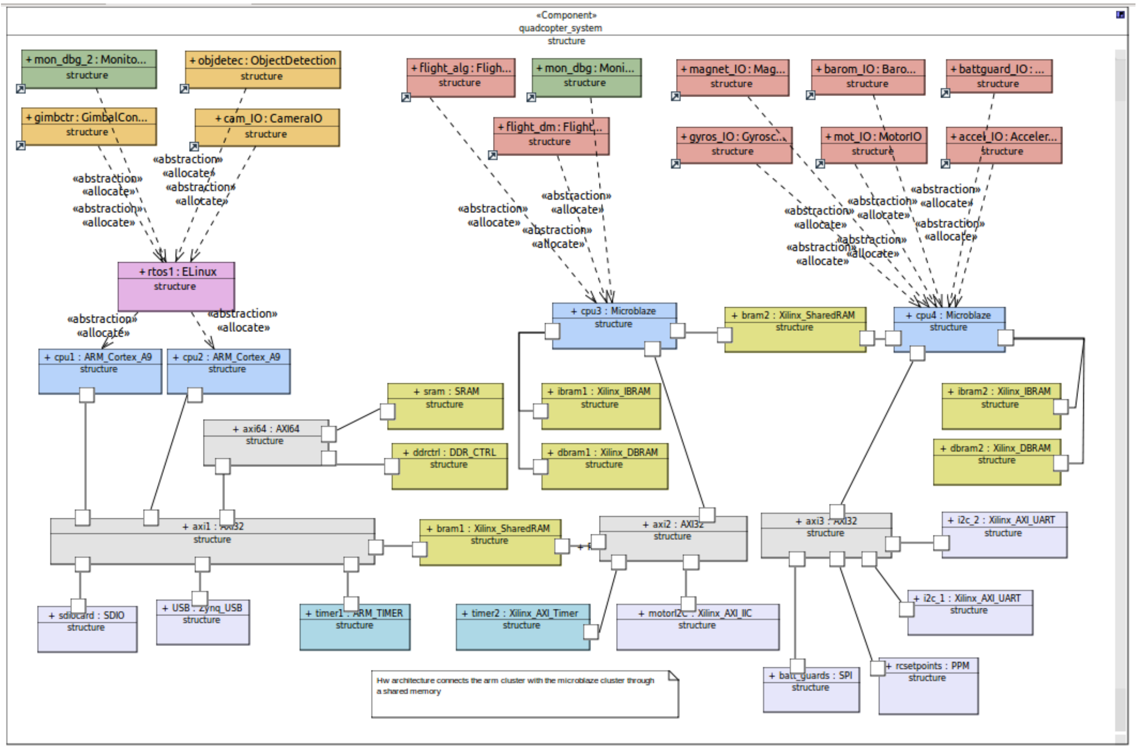


Figure 66 Mapping application component instances onto the HW/SW platform.

The destination of the mapping can be either a RTOS or a processing element.

There are a number of implicit assumptions and mapping rules. In general, a memory partition can be allocated to either one RTOS instance. It can be also allocated to a processing element. In such a case, there are two possibilities. If an OS instance has been allocated to such a processing element instance, then the mapping is equivalent (and thus a synthetic capture) to a mapping to that RTOS instance (with a specific affinity). However, if no OS instance is associated, it means a bare metal application.

Whe application component instances and schedulable resources are allocated to RTOS and processing elements similar rules apply. The mapping of the elements to different processing elements not associated to the same OS instance necessarily implies different memory partitions. A schedulable resource can be mapped to the several processing elements (meaning affinity in the case they are under the same RTOS).

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 67).

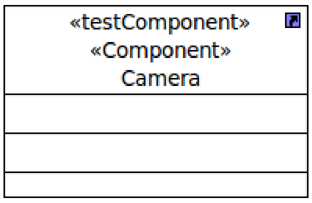


Figure 67 Environment component

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 68). The environment application components should be included in the *ApplicationView* like the rest of the application components of the system.

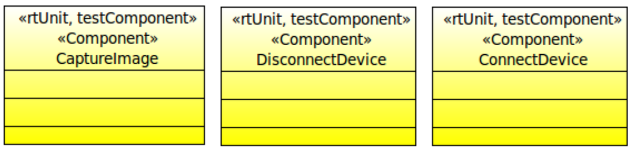


Figure 68 Environment application components

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 69).

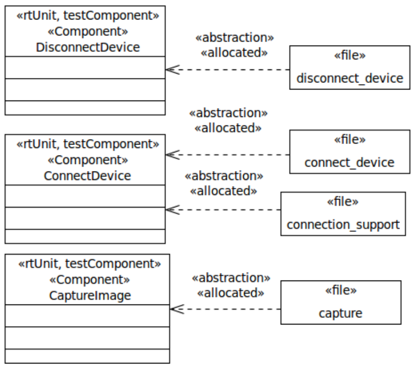


Figure 69 Environment Application components with associated Files

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 70) in a Composite structure diagram associated with the environment *TestComponent* component.

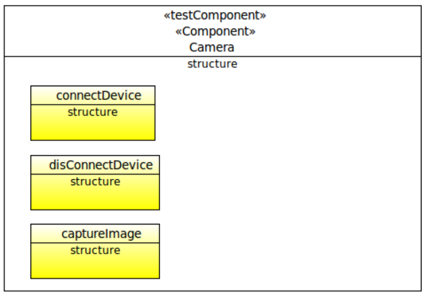


Figure 70 Application instances of an environment component

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 71 (Camera *TestComponent*). These *TestComponent* ports are connected to the internal application instance ports by using UML connectors (Figure 71). These application instance ports have to be named similarly to the *TestComponent* port that they are connected to (Figure 71).

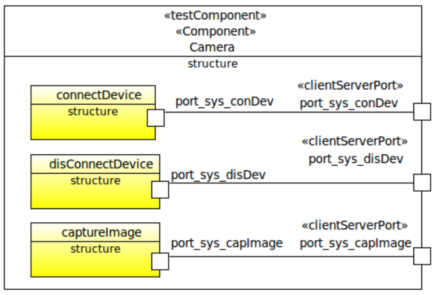


Figure 71 Environment Application components

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).

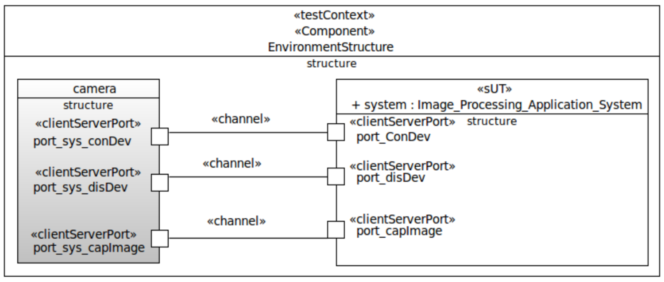


Figure 72 Definition of the environment structure

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 73).

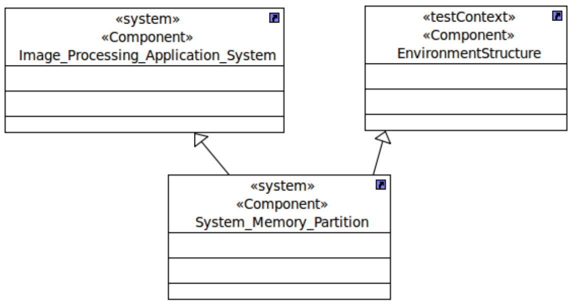


Figure 73 Generalization of Environment structure with the *System* component of the *MemorySpaceView*

Then, the allocation on memory spaces of the environment component (instances of *TestComponent* components) can be done (Figure 74).

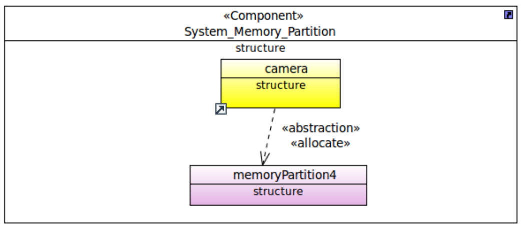


Figure 74 Allocation of environment component to the memory partitions

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.

11 Annex I: Methodology Stereotypes

Stereotype	**Attributes**	**Profile**
DataView		ESSYN
FunctionalView		ESSYN
ApplicationView		ESSYN
MemorySpaceView		ESSYN
HWResourcesView		ESSYN
SwResourcesView		ESSYN
ArchitecturalView		ESSYN
VerificationView		ESSYN
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]	ESSYN
File		Standard UML
ApplicationFile	parallelized: Boolean [1] highLevel: Boolean[1] implementation: String [0..1] notModifiable: Boolean [1] environment: Boolean [1]	ESSYN
SystemFile	systemConfiguration: Boolean [1] systemMetrics:Boolean[1]	ESSYN

	environment: Boolean [1] RTL: Boolean [1] TLM: Boolean [1]	
FilesFolder	parallelized: Boolean [1] highLevel: Boolean[1] implementation: String [0..1] notModifiable: Boolean [1] environment: Boolean [1]	ESSYN
ClientServerSpecification		MARTE
Pointer		ESSYN
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]	ESSYN
NotificationResource		MARTE
SharedDataComResource	identifierElements: TypedElement= [0..*]	MARTE
RtUnit	isMain : Boolean [1] main : Operation [0..*] srPoolSize: Integer [0..1] srPoolPolicy : PoolMgtPolicyKind [1]	MARTE
create		UML standard
Allocated		MARTE
ClientServerPort	kind : ClientServerKind [1]	MARTE

	provInterface : Interface [0..1] reqInterface : Interface [0..1]	
Channel	commType: CommunicationMedia [1]	ESSYN
System		ESSYN
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
HwISA	family: NFP_String [0..1] type: ISA_Type [1]	MARTE
Mode		MARTE
HwPowerState	frequency : NFP_Frequency [0..1]	MARTE

	Pstatic: NFP_Power [0..1]	
ModeTransition		MARTE
HwPowerState Transition	setUp : NFP_Duration [0..1]	MARTE
ResourceUsage	powerPeak : NFP_Power [0..1]	MARTE
OS	type:String [1] scheduler: Scheduler[*] drivers: DeviceBroker [*] interProcessCommunication: InterProcessCommunicationMechanism [1]	ESSYN
DeviceBroker		MARTE
TestComponent		UTP
TestContext		UTP
SUT		UTP
Reference		ESSYN
Qualifier	qualifier:Qualifier [1]	ESSYN

Table 8 List of Stereotyes and attributes used in this modelling methodology.

12 Annexo II: 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 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	ESSYN
Qualifier	None	ESSYN

	Const Volatile register	
PollMgtPolicyKind	infiniteWait timedWait dynamic exception other	MARTE
ClientServerKind	proreq provided required	MARTE
CacheType	data instruction unified	MARTE
ISA_Type	RISC CISC VLIW SIMD Other Undef	MARTE
CommunicationEngineKind	undef default MCAPI OPenMP OpenStream TCP/IP	ESSYN
CommunicationOSServiceKind	undef FIFO Socket messgeQueue SharedMemory File	ESSYN
InterProcessCommunicationMechanism	FIFO	ESSYN

	Socket MessageQueue SharedMemory File	