UML/MARTE Methodology for Synthesis

April, 2015

Microelectronics Engineering Group
TEISA Dpt. , University of Cantabria
Authors: P. Peñil

Index:

Index of Figures:

1 Introduction

The UML/MARTE methodology enables to establish a synthesis design flow that, taking as starting point, the SW to be executed in a specific HW/SW platform can be done. For that purpose, the UML/MARTE model should include additional information. This information is related to compilers, compilation and link flags, files for specific HW resources… In that way, a toolkit can obtain all the required SW infrastructure (makefiles, SW of deployment) for the system implementation in a target board.

2 Model specification

2.1 Data Size

All the data includding the *DataView* modelling must include the size in bytes. This value is captured in the attribute size of the stereotype <<DataSpecification>>.

2.2 Refinement of files

Two different kinds of File artifacts can be defined in the *FunctionalView*: the artifacts only specified by the stereotype <<File>> and the artifacts specified by both stereotypes, <<File>> and <<ApplicationFile>>. In the first case, these files represent the functionality provided in the initial stage of the design flow. The combination of the stereotypes <<File>> and <<ApplicationFile>> means that the functionality of the corresponding artifacts has been refined for executing on a specific HW resource or that it has been modified by an external tool or by the user. In addition, the latter *files* can represent different file structures used for the different stages of the design process. In any case, the model should capture the relationship between the initial *files* and the refined *files*. This *file* refinement is captured by a UML Abstraction relationship between a *file* with a set of files. This UML abstraction is specified by the UML standard stereotype <<refine>>, as can be seen in Figure 1. Only one refined file is allowed for each design stage. There is one exception; when two files contain optimized code for two different, specific HW resource. For instance, two different implementations, one for a NEON execution and other one for a DSP are shown in Figure 1. Depending on the HW resource where the application is mapped, the code generation annotates the correct file.

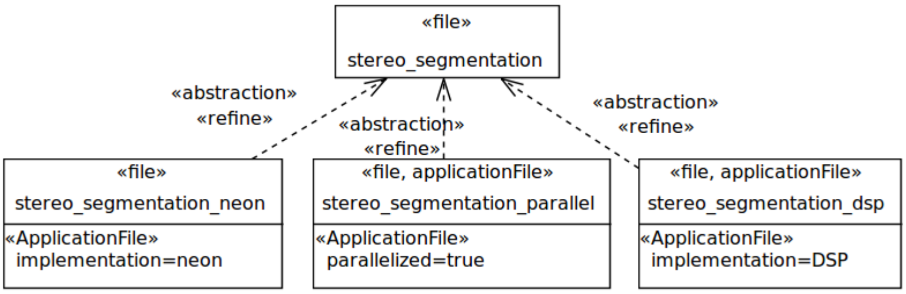

Figure 1 Refinement of Files

2.3 Channels

The channels have information about the way they should be implemented. This information is captured in attributes associated to the stereotype <<Channel>>. The attribute *communicationEngine* is an enumeration with a set of communication libraries independent of the platform. The possible values are *MCAPI*, *OpenMP*, *OPenStream*, *TCP/IP* and *default are;*

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

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

When the values of the attributes *communicationEngine* and *communicationOSService* are *undef* and *default* respectively, it means the communication mechanism implemented for a channel derives from the *OS* where the interconnected application components are mapped. The attribute that defines this implementation mechanism is *interProcessCommunication*.

2.4 Application components: compiler, flags and APIs

The application structure can have aasocited information for enabling the compilation and generation of the executable code, abstracting a specific HW/SW platform captured in the *ArquitecturalView*. For that purpose, additional modelling variables should be considered:

1. *cc_compiler*: specifies the C compiler.
2. *cxx_compiler:* specifies the C++ compiler.

3. *path_compiler:* specifies the path where the compiler (C or C++) is allocated.
4. *CFLAG*: defines the compilation flags
5. *LFLAG*: defines the linking flags.
6. *ImplementationAPI*: denotes which API should be used in the sysnthesis process for implementing the component.

2.4.1 System component

The *System* componente of the *ApplicationView* can have can have associated all the previous modelling variables.

CFLAGS and LFLAGS

The model variables associated with the *System* component of the *ApplicationView* can include the set of CFLAGs and LFLAGS required for the native compilation of the application (Figure 2).

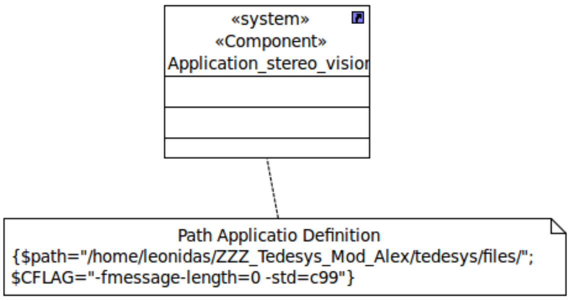

Figure 2 $CFLAGs for native compilation

Compiler and Compiler path

The model variables associated with the *System* component of the *ApplicationView* can include the compiler (for C or C++) required for native compilation and the path where this compiler is allocated (Figure 3). By default, gcc and g++ are the compilers considered for compilation.

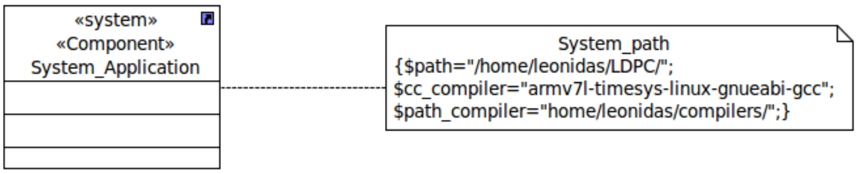

Figure 3 Compiler variable

2.4.2 Aplication component

The application componets can have associated the modeling variables CFLAG and LFLAG. In that way, the designer can captured specific flags for a specific component that are added to the flags asocited to the *System* component.

In addition to that, a specific API for its synthesis implementation can specify for the application instancies. The modelling variable *$implementationAPI* is used for that purpose; APIs as OpenMP and MCAPI. In the case this modelling variable is not specifed, a default API is used, which is POSIX.

The variables are annotated in a UML constraint that is owned by the component where the application instance is created; in the *System* component or in a *Subsystem* component. The, the UML constraint is associated to the application instance by a link as in the previous examples.

2.5 HW Processor variables

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

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

These variables are specified in a UML constraint (Figure 4). This constraint is owned by the HW Processor (the attribute "Context" has to contain the HWProcessor component to be constrained) and associated with a HwProcessor component by uisng a UML link.

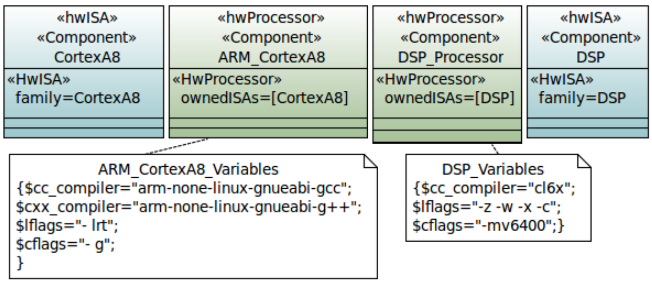

Figure 4 HwProcessor compilers

2.5.1 DSP processors

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

2.5.1.1 Allocation on DSP

When the the memory allocation is done on a DSP, the allocation is captured by means of a UML abstraction specified by the MARTE stereotype <<Allocate>>. However, the mapping is captured directly from *MemoryPartition* instance to the DSP resource, without any *OS* in the middle (Figure 5).

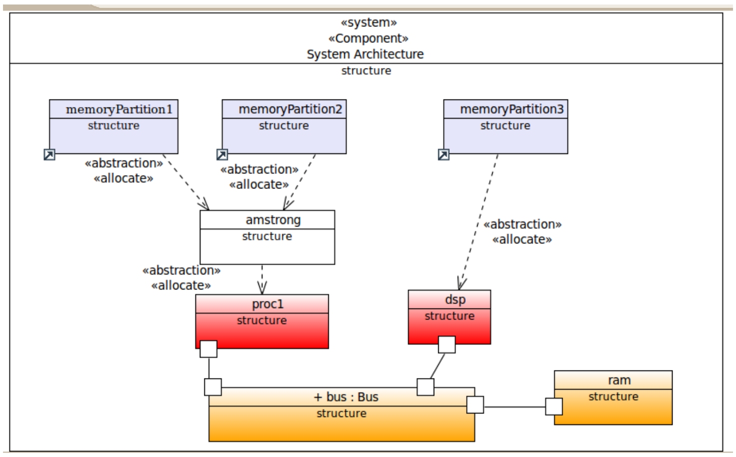

Figure 5 Memory partition allocations to DSP

The memory partition instance mapped onto the DSP HW resource has a modelling restriction; only one application component can be allocated to a memory partition that is mapped onto a DSP (Figure 5 and Figure 9).

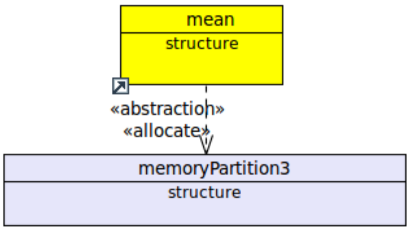

Figure 6 Application component allocation to a memory partition

2.5.2 GPU processors

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

2.5.2.1 Application Allocation to GPU

The application components are mapped onto memory partitions and then, these memory partitions are mapped onto HW/SW resources of the platform. A special case of application mapping is the mapping onto GPU HW resources.

In this specific case, the element mapped on the GPU resource is the application instance as Figure 7 shows.

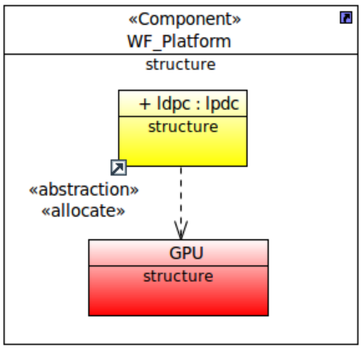

Figure 7 Application component instance for GPU mapping

2.5.3 CPU co-processors

CPUs may have associated co-processors which may affect the compilation process. So, the "CortexA" processor has an associated NEON co-processor (www.arm.com/products/processors/technologies/neon.php). In the case that a *HwProcessor* has an associated *HwISA* specified as "CortexA?" (where the "?" represents any possible value, Figure 8), the eclipse plug-in generates the entire infrastructure for using the NEON co-processor to execute functionality. The designer can select which application components should be executed in the NEON co-processor.

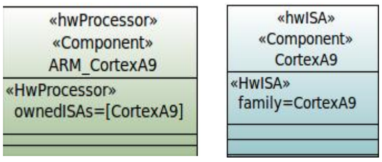

Figure 8 HW Specification of a CortexA processor

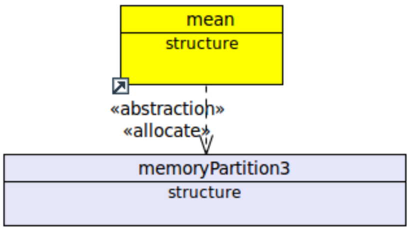

Figure 9 Application component allocation to a memory partition

2.5.3.1 Processor identifier

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

2.6 Multiple HW resources allocation

The modelling methodology enables multiple allocations of the memory spaces in different HW resources of the platform as can ben seen in Figure 10.

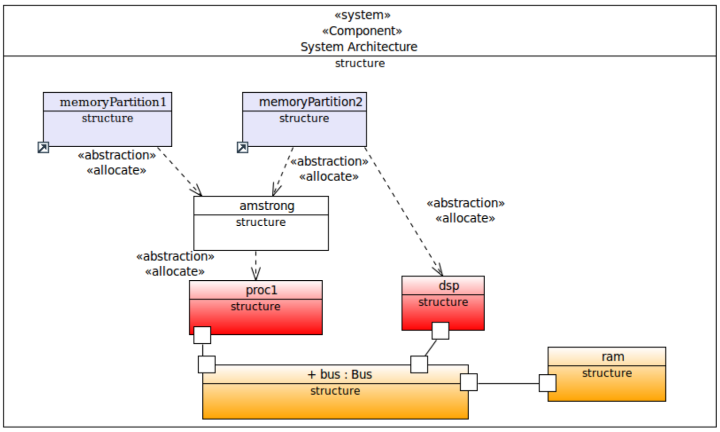

Figure 10 Multi HwResources allocation

From, these multiple allocations, the adequcete code is synthesized in order to enable the execution on both HW resources.

3 Annex I: Methodology Stereotypes

Stereotype	Attributes	Profile
DataSpecification	size:NFP_Data [1]	ESSYN
Channel	communicationEngine: CommunicationEngineKind[1] communicationOSService: communicationOSServiceKind [1]	ESSYN
ApplicationFile	implementation: String [0..1]	ESSYN
OS	interProcessCommunication: InterProcessCommunicationMechanism [1]	ESSYN
Refine		UML Standard

4 Annexo II: Methodology Enumerations

Enumeration	Values	Profile
CommunicationEngineKind	undef default MCAPI OPenMP OpenStream TCP/IP	ESSYN
CommunicationOSServiceKind	undef FIFO Socket messgeQueue SharedMemory File	ESSYN
InterProcessCommunicationMechanism	FIFO Socket	ESSYN

	MessageQueue SharedMemory File	