

Metamodels and MDA Transformations for Embedded Systems

Abstract

Embedded system design needs to model together application and hardware architecture. For that a huge number of models are available, each one proposing its own abstraction level associated to its own software platform for simulation or synthesis. To produce a co-design framework, we are obviously obliged to support different models among all possible ones. Between these models we should produce automatic transformations. Each time a new model is included in the framework, we should develop a new transformation.

To improve transformation engine development, Model Driven Architecture (MDA) techniques are useful. This approach permits to define the transformations at the metamodel level. It guaranties to the framework the reuse of models and unifies the definition of the transformation rules.

We present the application of MDA in the context of Intensive Signal Processing (ISP) applications deployed on System on Chip (SoC) platforms. For that purpose, we have developed a new MDA Transformation engine: `ModTransf`. We apply this engine on UML profiles to generate SystemC Transaction Level Model dedicated to ISP. A particular rule will be presented to illustrate the interest of this approach in a multi model embedded system design environment.

1 Introduction

The MDA is based on models describing the systems to be built. A system description is made of numerous models, each model representing a different level of abstraction. The modeled system can be deployed on one or more platforms via model to model transformations. A key point of the MDA is the transformation between models. The transformations allow to go from one model at a given abstraction level to another model at another level, and to keep the different models synchronized. Related models are described by their metamodels, on which we can define some mapping rules describing how concepts from one metamodel are to be mapped on the concepts of the other metamodel. From these mapping rules we deduce the transformations between any models conforming to the metamodels. The MDA model to model transformation is in a standardization process at the OMG [Obj03a]

The MDA is based on proven standards: UML for modeling and the MOF for metamodel expression. The new coming UML 2.0 [Obj03b] is specifically designed to be used with the MDA. It removes some ambiguities found in its predecessors (UML 1.x), allows more precise descriptions and opens the road to automatic exploitation of models. The MOF (Meta Object Facilities [Obj00]) is oriented to the metamodel specifications.

Our proposal is partially based upon the concepts of the `Y-chart` [GK83]: application, hardware architecture and then association to map one application on one hardware architecture. The MDA contributes to express the model transformations which correspond to successive refinements between the abstraction levels, from PIM to PSM. In this paper we present the transformation of the association PIM to SystemC PSM for a System on Chip design. For this transformation we use the tool `ModTransf` developed in our research group in respect of the QVT proposal.

2 The Transformation Engine : ModTransf

Model to model transformations are at the heart of the MDA approach. Anyone wishing to use MDA in its projects is soon or later facing the question: how to perform the model transformations? There are not so much publicly and freely available tools, and the OMG QVT standardization process is not completed today. To fulfil our needs in model transformations, we have developed **ModTransf**, a simple but powerful transformation engine. **ModTransf** was developed based on the recommendations done after the review of the first QVT proposals [TCKR03]. Based on these recommendations and on our needs, we have identified the following requirements for the transformation engine:

- Multi models as inputs and outputs
- Different kind of models: MOF and JMI based, XML with schema based, graph of objects
- Simple to use
- Easy modification of rules to follow metamodel changes
- Hybrids: Imperative and declarative rules
- Inheritance for the rules
- Reversible rules when possible
- Customizable, to do experimentations
- Free and Open-Sources.

The proposed solution fulfil all these needs: **ModTransf** is a rule based engine taking one or more models as inputs and producing one or more models as outputs. The rules can be expressed using an XML syntax and can be declarative as well as imperative. A transformation is done by submitting a concept to the engine. The engine then searches the more appropriate transformation rule for this concept and applies it to produce the corresponding result concept. The rule describes how properties of the input concept should be mapped, after a transformation, to the properties of the output concept.

2.1 Basic Principle

Transforming a model can be a very complex task. **ModTransf** helps to reduce this complexity by allowing the specification of a model transformation *rule by rule*. A rule specifies how to transform one or few input concepts to one or few output concepts. This *divide and conquer* approach allows focusing on simple transformations, improves the readability, open the road to rule inheritance and eases the maintenance. In the XML rule language, a rule specifies the source concepts it uses, the concepts it produces in the destination model, and which properties of the source concepts are used to populate the properties of the destination concepts. The rule does not specify how to transform these properties; it only specifies which properties must be transformed and where to store the result. It is the engine responsibility to search and execute the more appropriate rule.

This way of expressing the rules induces recursive calls to the engine, and provides a natural scan of the model to transform. By default a rule is called only once for a given set of inputs. Subsequent calls will return the same results than the first call. This allows breaking the recursivity and avoids multiple transformations of an object: if a source object is referenced by several properties, it will be transformed only once.

The transformation of a model or an object is performed by submitting it to the engine. The engine looks for the most appropriate rule which in turn call the engine to transform the child objects. Thus the entire graph of objects associated to the first object is transformed by using the most appropriate rule for each node of the graph.

The rules can be organized in *rule sets* used as search unit by the engine. It is then possible to specify which rule set should be used for a transformation. If no rule set is specified, the current one is used by default. Rule sets can be used to reduce the scope of a search or to provide several rules transforming the same input concepts, but used in different contexts. It is also possible to specify explicitly which rule should be used by the engine. In this case, the transformation is imperative and the engine uses directly the rule without performing any search.

The rules are searched in the order of their declarations in the rule set. By default, only the first matching rule is executed. This behaviour can be changed by specifying that all matching rules should be executed. Input and output models are submitted to the engine as graphs of objects. The engine and the rules access to the graph of a model through a well known API defining the basic methods they need: attribute access, concept creation, ...

The access API allows making use of different technologies to manipulate the models: JMI and its different implementations (MDR, NsUML, CIM, ModFact, ...); EMF; DOM representation of XML files; models generated from XML schema or DTD with tools like Jaxb, Castor; or any kind of object graph. An implementation of the API is linked to the particular technology used to represent models. Generally it should be developed only once for this technology.

The rules understood by the engine are very simple: they are made of only one guard and one action. The guard is evaluated to select the rule, and the action is executed when the rule is selected. More complicated rules can be built on top of this basic interface. Thus it is possible to write rules directly in Java, as well as in a dedicated high level language with a dedicated interpreter or compiler. One can develop its language and compiler or interpreter. To avoid the burden of such development, we propose a language written using XML, and allowing complex rules.

2.2 The XML Rules

Rules defined in XML use a concept of left and right instead of source and destination. This does not presuppose on the direction of the transformation, allowing writing transformation rules potentially reversible.

The left and right notions will be translated to sources and destinations according to the direction of the transformation. If the transformation flows from left to right, the left will become the source and the right the destination. If the transformation is performed in the other direction, right becomes the source and left the destination.

The reversibility of the rules is possible only in certain cases. The complete transformation is reversible only if all the rules are reversible. Actually, this feature is only for experimentation. In the remaining of this paper we suppose that the transformation flows from left to right.

An XML rule is made of left conditions, right conditions, and actions. The conditions are used to describe the source concepts used by the rule, and the concepts that the rule should produce. The description of a condition is the same though it is used as source or as destination: it generally specifies the type of the concept, and optionally the conditions expected on some properties of the concepts or on any sub-properties. To ease the transformation of models described in UML with an associated profile, some dedicated conditions are provided, like testing or setting a stereotype or a tagged value.

When the transformation is performed, the source condition becomes a guard testing the concept while the destination condition becomes an action creating the expected concept. The actions are used to populate the properties of the destinations concepts from the properties of

the source concepts, with eventually a transformation. Two main actions are used: The first copy primitive types, with eventually a conversion between the types; the second specifies one or few properties of the source concept that should be transformed to one or few properties of the destination concept by calling the transformation engine to select the more appropriate rule.

Action arguments are specified by using *accessors* allowing to express the source and the destination arguments in exactly the same way. Different kind of accessors are provided to access to a concept, a property, a tagged value, ...

The XML language provides basic conditions, actions and accessors that should cover current needs in model transformations. Should you encounter a special need that is not cover yet, or should you want to propose a special behavior to simplify your transformation, you can provide your own condition, action, accessor or even complete rule. This is done by implementing a Java class providing the desired behavior. Likely all elements of the language accept customized behavior in place of the default behavior.

The `ModTransf` engine is an Open Source project available on the net [Dum03]. We will now see how it is used in our Embedded Application for Soc Design project.

3 PIM and PSM metamodels for embedded systems

We propose a construction of metamodels to support a co-design methodology [DBDM03]. This proposal is partially based on the concept of Y-chart. We have defined and formalized our concepts in MOF and these MOF specifications have been implemented in UML profiles using Tau G2.

In our approach we design a system starting from two initial models: the application part defines functions and services provided by the system and the hardware architecture part represents an abstraction of the hardware material on which the application will be executed. These two models are then mapped to make the association model. This latter expresses associations between the functional components and the hardware components. Each of these three models are instances of their corresponding metamodels, they are Platform Independent Models. To realize a simulation of this embedded system, we propose the Platform Specific Model for Transaction Level Simulation in SystemC. `ModTransf` will produce the transformation from PIM association to PSM SystemC.

Our metamodels are too large to be exhaustively presented in this article. We will therefore give an overview of the main concepts, leaving out details.

3.1 The application metamodel

The application metamodel is defined in the ISP-UML profile. It is based on the Array-OL language (Array Oriented Language) designed by Thomson Marconi Sonar and dedicated to Intensive Signal Processing. Array-OL introduce the notions of local model and global model. In the ISP-UML metamodel, we propose a set of concepts to specify the application part of a system. An application is described by assembling component wich can be of three types: **CompoundComponent**, **DataParallelComponent**, and **ElementaryComponent**.

- A **CompoundComponent** can contains other sub-components. It expresses the global model of Array-OL and shows component interconnections.
- The **DataParallelComponent** is the heart of intensive signal processing (similar to the local model of Array-OL). It is made of a unique nested component (eventually an **ElementaryComponent**) and of one tiler component for each of its connections. The tilers are used to describe how the data are spraid among the instances of the nested component.

- The **ElementaryComponent** directly refers to an external implementation. It is a computation unit which has no further detailed structure. It gets its input data from input tilers and the result of the computation is carried out through an output tiler.

In the metamodel a **Tiler** instance is represented by an **AolTilerPart** which is introduced to add some tagged values necessary to specify the origin, the fitting and the paving vectors. **Ports** represent proxies for data handled by components. They are used as endpoints of connections. A port specifies the type of data it carries, itself defined by an interface in the Object Oriented sense. In ISP UML, all AOL arrays are defined by inheriting from an interface called **AolArray** providing basic attributes: element type, number of dimensions and size of each dimension.

A broad description of the application metamodel is given in figure 1.

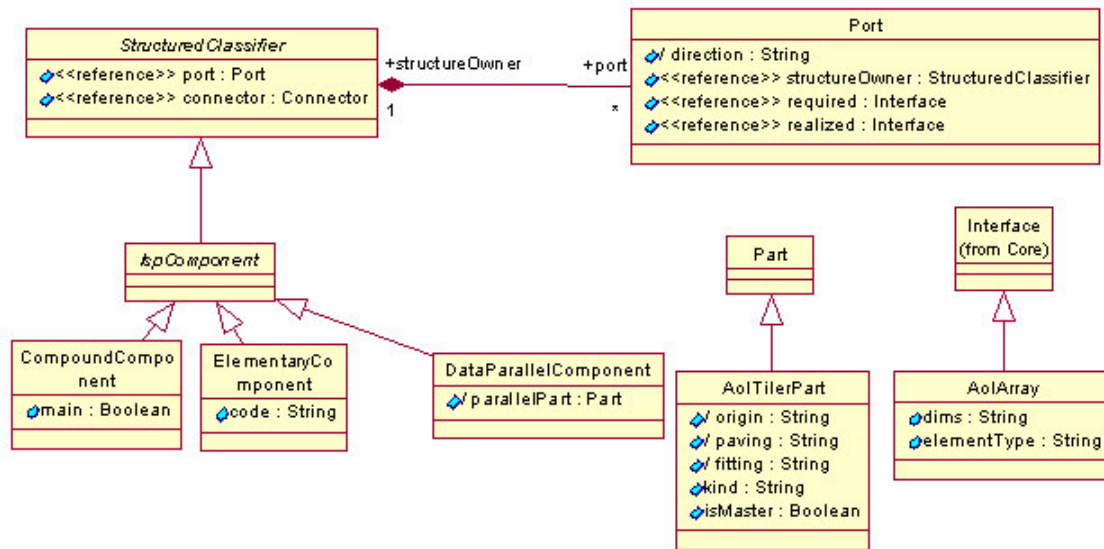


Figure 1: Overview of the application metamodel

3.2 The hardware architecture metamodel

The architecture metamodel is similar to the ISP-UML metamodel: it provides components to describe a hardware architecture. The overview of the metamodel is shown on figure 2. The hardware component represents abstractions of physical hardware architecture elements. We propose to classify the resources according to two criteria: a functional and a structural (see figure 2).

- Structural concepts : **ElementaryHwComponent**, **CompoundHwComponent** and **RepetitionHwComponent** are used to describe the structural aspect of the architecture.
 - The **ElementaryHwComponent** is a component without an internal structural description. For example, it could be used in the case where we have an hardware IP for this component, or in the case where we don't want to model the component more finely.
 - The **CompoundHwComponent** is a component with an internal structure description. The interest in using such a concept is to provide a means for hiding details that are not necessary at a certain level of specification, and also the reuse of existing blocks in modelling other architectures.

- The **RepetitionHwComponent** is a kind of particular **CompoundHwComponent**, which contains a repetition of the same hardware component. This kind of component is well suited to the modelling of massively parallel architectures and is motivated by the recent introduction of such architectures in the design of SoC such as the picoChip PC101 and PC102 [pic03].
- Functional concepts : **PassiveHwComponent** , **ActiveHwComponent** and **InterconnectHwComponent** are used to specify the functions of the architecture elements.
 - The **PassiveHwComponent** is a storage unit. It stores the data of the application. We typically use it as a representation of elements such as RAMs, ROMs, sensors, or actuators.
 - The **ActiveHwComponent** is a processing unit, it reads or writes into passive resources. This category includes CPUs, DMAs or SMP nodes inside a parallel machine.
 - The **InterconnectHwComponent** is a hardware unit used to specify connections between active and passive components or active and active in the case of a distributed memory architecture. This category includes elements such as bus or an interconnection network.

Each hardware component should be tagged with these two aspects, each one representing a different view on the component. This lead to 9 possible types of hardware components.

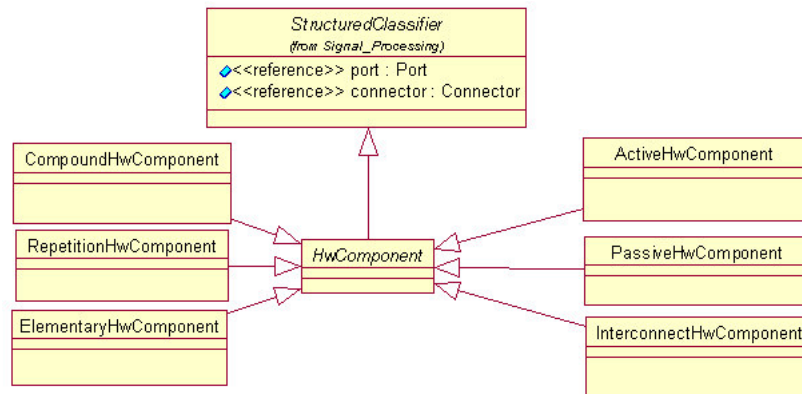


Figure 2: Overview of the architecture metamodel

3.3 The association metamodel

The aim of the association model is to point out where each software component will be executed, the location of the data used by the software, and the channels used for the communication between the different hardware components. It is also intended to show how the different executable components are scheduled. This scheduling is static, and decided at the association level. In other words the association metamodel defines a mapping of the application specification and the architecture specification. Therefore the association metamodel imports the application and architecture metamodels and adds to them the following concepts: **CodeMapping**, **DataMapping** and **Scheduling**.

The overall view of the metamodel is given in figure 3. It is important to note that the **Part** concept stands for an instance of either a application component or an hardware component.

The **CodeMapping** specifies on which hardware components the different software components are assigned. The *runables* attribute references software components that are executed

on the hardware component referenced by the *activeComponent* attribute which practically is a reference to a processor (or a group of processor).

The **DataMapping** concept specifies where the application data (arrays) are placed in the architecture. It is mainly the specification of the mappings of the data (arrays) on the memory.

The **Scheduling** concept is used to define the order in which components are processed in the case of several application components assigned to the same architecture unit. This scheduling is local to each hardware component.

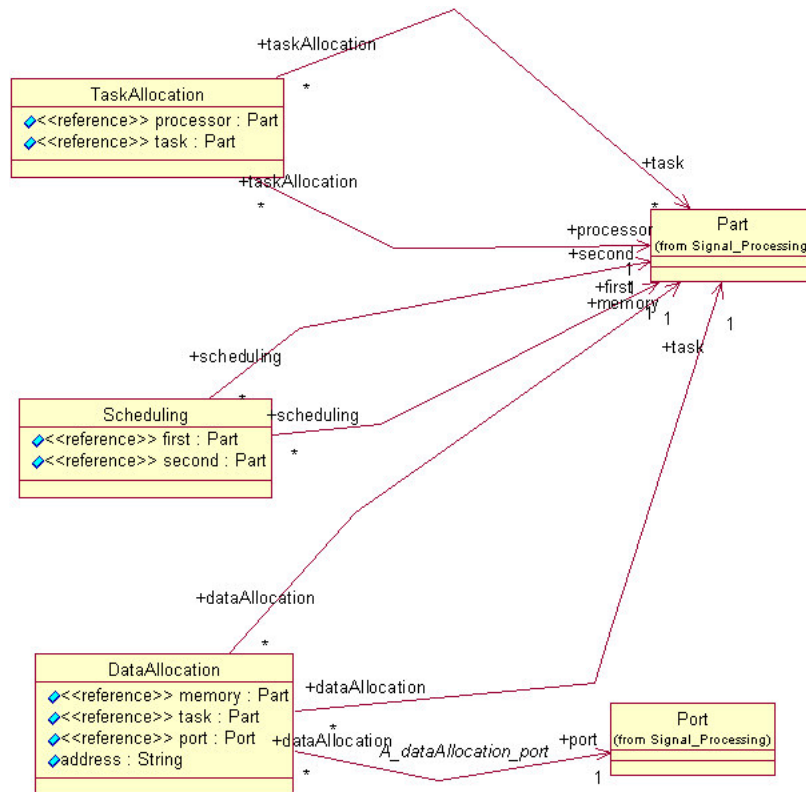


Figure 3: Association metamodel.

3.4 The System C metamodel

SystemC is a C++ class library and a simulation kernel for hardware, software and system modeling. It is particularly suited to:

- propose a methodology for SoC designs consisting of DSPs, ASICs, IP-Cores, Interfaces, ...
- extend C/C++ by providing concurrency, timing, reactivity, communication, signal/data types,...
- simulate up to the level of cycle-accurate.

Like any C++ program, a SystemC application contains a *Main* function which depends on a set of *Modules*. The module concept defines the hierarchy of instanciable modules. The metamodel presented here is not a definition of the SystemC library. It is rather a metamodel oriented towards SystemC code generation for the particular case of Intensive Signal Processing mapped on a SoC. The metamodel is provided in figures 4 and 5, its main concepts are : *Main*, *Module* and *ModuleInstance*.

- The **Main**

It is the top level component of a SystemC specification. It holds a set of module instances and signals to model communications between these instances.

- The **Module**

It is a pure SystemC concept. It is subclassified in **TaskModule** and **HwModule**(see figure 5) in order to take into account the application and the architecture parts of the system.

- The **ModuleInstance**

The **ModuleInstance** is an instance of a module. Since we consider a **Module** to be a Class specification, the **ModuleInstance** is therefore an instance of the class **Module**, in other words, it is an object. The **ModuleInstance** is an abstract class subclassified in **TaskModuleInstance** and **HwModuleInstance** (see figure 6).

In the code generation step, each Module leads to the generation of a file, while the ModuleInstance is used to create attributes and instances.

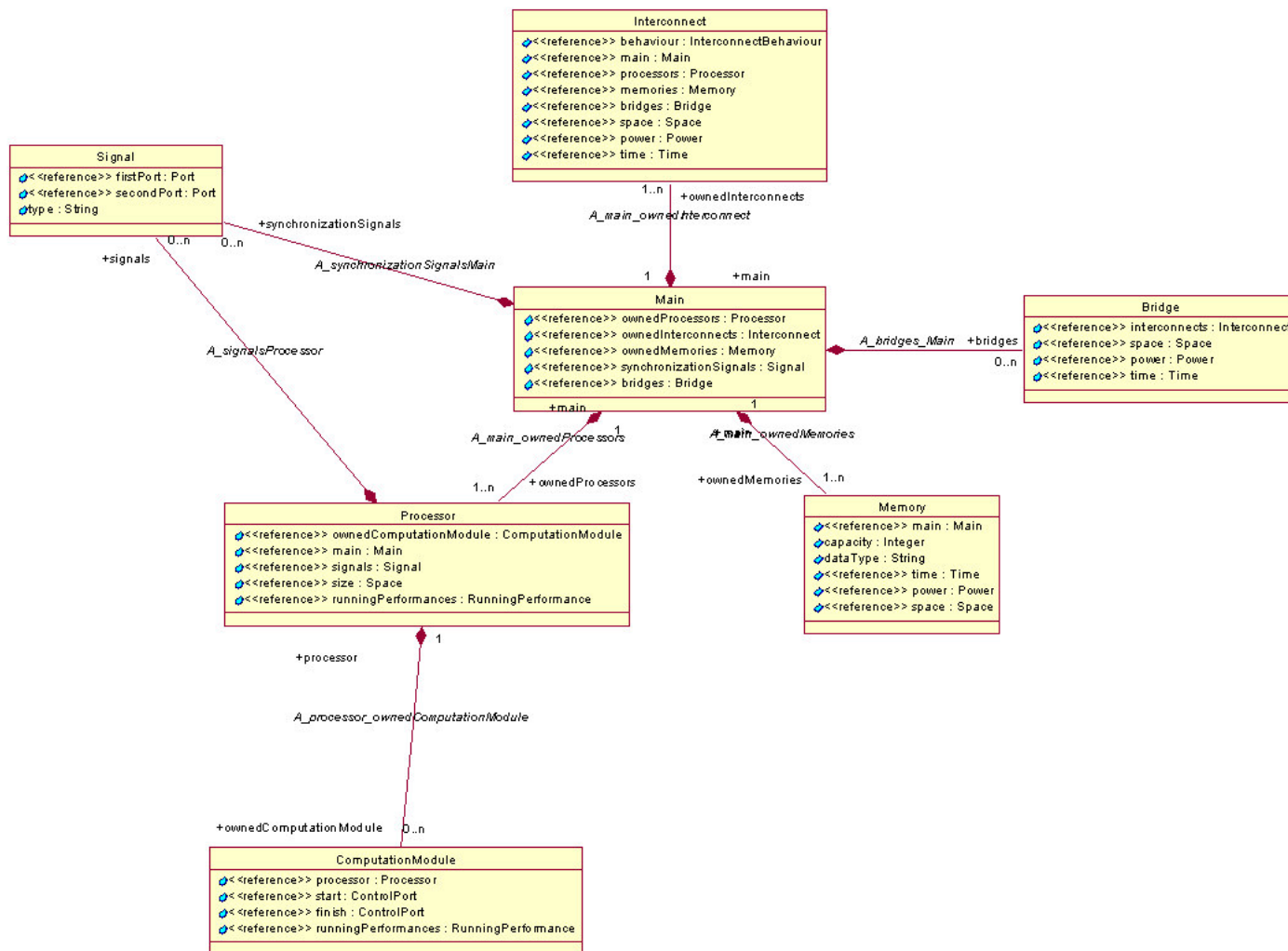


Figure 4: System C simulation metamodel : Overview

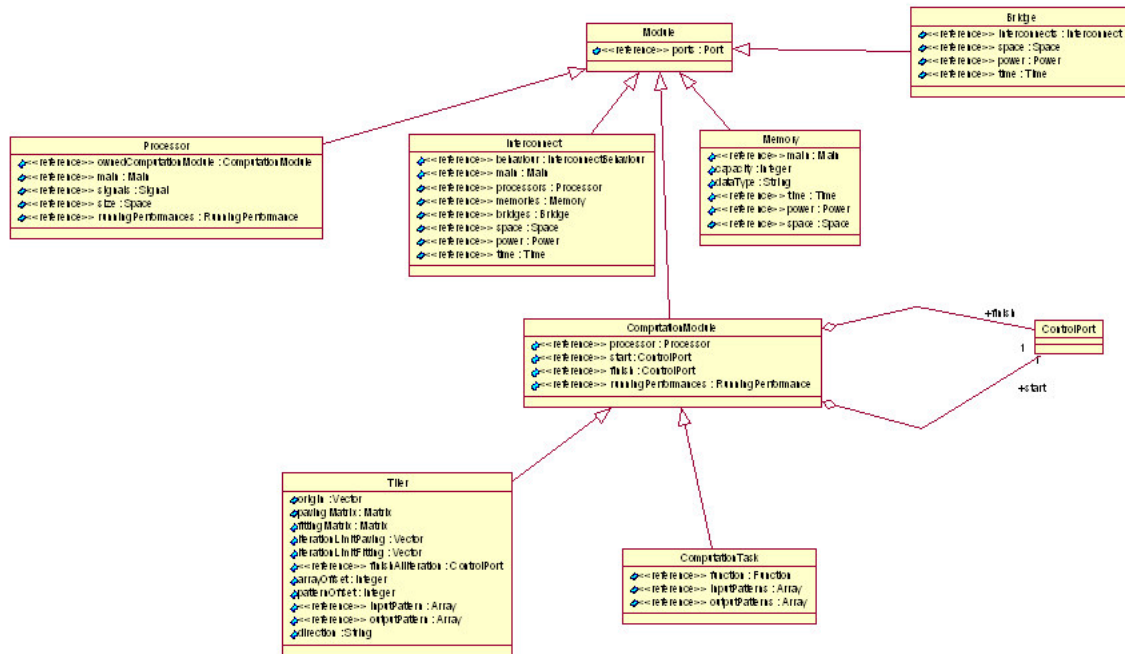


Figure 5: System C simulation metamodel : Module

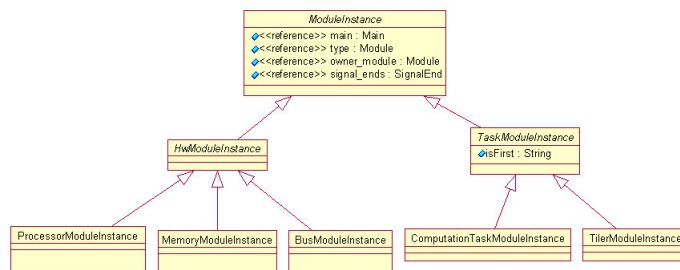


Figure 6: System C simulation metamodel : Module Instance

4 PIM transformation to PSM

The transformation of our PIM *association* metamodel to our PSM *SystemC TLM simulation* metamodel requires the development of a set of dedicated mapping rules. This development requires the identification of the main mapping rules, and then the detail of each of these mappings. In this section we will show the main rules of our transformation, and the details of one of these rules. Then we will explain the implementation of the selected rule with `ModTransf`.

4.1 Main Mapping Rules

The `ModTransf` tool allows a development rule by rule, where, ideally, each rule focus on a simple concept to concept transformation (even if the tool supports a one to one, one to many, and many to one mappings). Thus a complete transformation is made of basic rules. Our transformation main rules mapping concept from the association metamodel to concepts of our SystemC metamodel are given in the following table:

Association metamodel concepts		SystemC metamodel concepts
CompoundComponent		TaskModule
DataParallelComponent		TaskModule
ElementaryComponent		ComputationTaskModule
Part of type DataParallelComponent		TaskModuleInstance
Part of type ElementaryComponent		ComputationTaskModuleInstance
AolTilerPart		TilerModuleInstance
CompoundHwComponent RepetitionHwComponent	+ ActiveHwComponent	ProcessorModule
CompoundHwComponent RepetitionHwComponent	+ PassiveHwComponent	MemoryModule
CompoundHwComponent RepetitionHwComponent	+ InterconnectHwComponent	BusModule
ElementaryHwComponent + ActiveHwComponent		ProcessorModuleInstance
ElementaryHwComponent + PassiveHwComponent		MemoryModuleInstance
ElementaryHwComponent + InterconnectHwComponent		BusModuleInstance
CodeMapping		CodeMapping
DataMapping		DataMapping
Scheduling		Scheduling
Port		Port
Connector		Signal

This top level mapping is not sufficient. We need now to detail each rule to specify how properties of a source concept map to properties of the destination concept. Each attribute or feature will be either copied (for simple data types) or transformed from the source concept to the target one. Let us consider for example the mapping between `DataParallelComponent` and `TaskModule`. The details of the mapping are given in the table below:

DataParallelComponent attributes	Operation(action)	TaskModule attributes
name : String	Copied	name : String
ports : Port[*]	Transformed	ports :Port [*]
connector : Connector[*]	Transformed	signals : Signals[*]
feature : [*]	Transformed	ownedInstances : [*]

These transformations look like a direct and simple one to one transformations. But it is not actually the case. For example in the transformation

`feature : [*] ---> ownedInstances : [*]`

each element in the collection *feature* are not necessary of the same type. They will be transformed individually using the appropriate rule, and the result will be added to the *ownedInstances* collection attribute. In our example, the *feature* attribute contains several *AolTilerPart*, and a *part* of type *ElementaryComponent*, *DataParallelComponent* or *CompoundComponent*.

Once the concept mapping and the details of the mappings are defined, the next step is to implement them using `ModTransf`.

4.2 Transformation Rule in ModTransf

In this example, we transform a **DataParallelComponent** concept from the association meta-model to a **TaskModule** in the SystemC simulation meta-model. Figure 7 shows graphically this transformation.

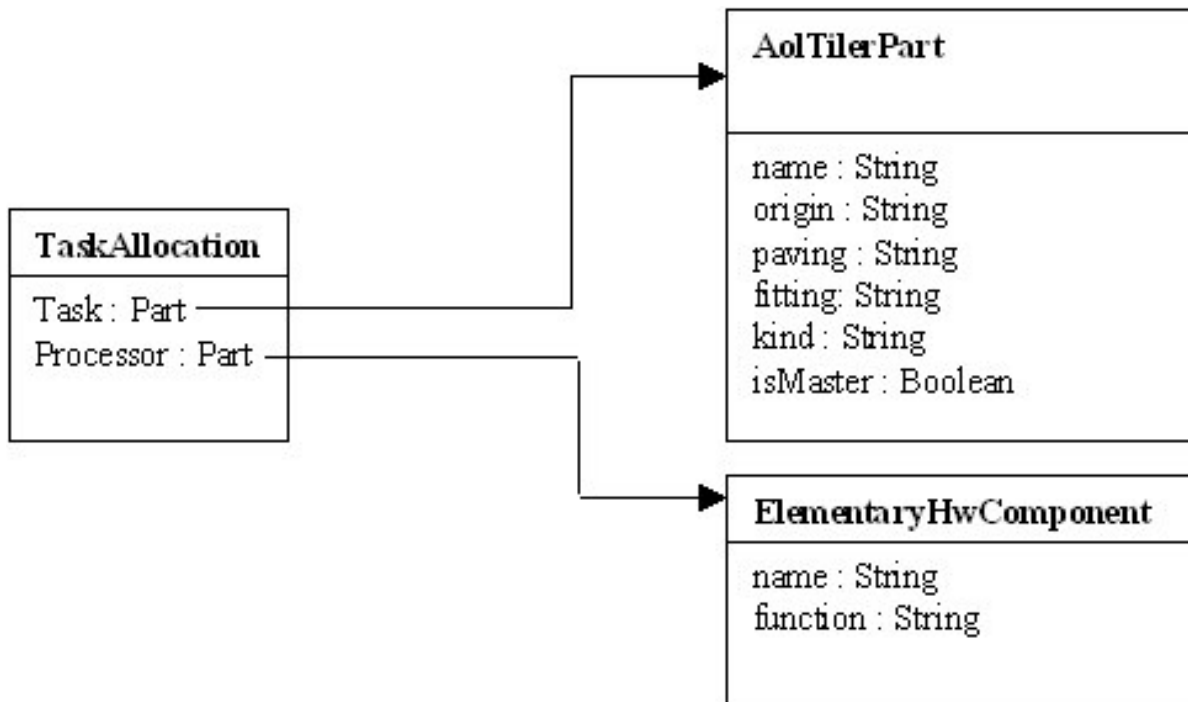


Figure 7: Transformation of a DataParallelComponent into a TaskModule

The XML rule of `ModTransf` start by specifying the left and the right concepts involved in the transformation. Here, we specify that the rule takes a **DataParallelComponent** as input and produces a **TaskModule** as output. Then, the rules describe the list of actions needed to transform the properties of the left concept to properties of the right concept, as follows:

- The *name* property value will be assigned to the *name* property of the output object.
- The *ports* property is a collection; each element in that collection will be transformed and the result will be added in the elements of the *ports* property (which is also a collection).
- Each element in the *connector* property collection will be transformed and the result will be added to the collection property *signals*.
- And finally each elements of *feature* will be transformed and added to the *ownedInstances* property.

The xml Rule code is as follows:

```
<rule ruleName="DataParallelComponent">
  <description> Transform a DataParallel Component </description>
  <leftConditions>
    <concept type="Signal_Processing.DataParallelComponent" model="isp" use="required"/>
  </leftConditions>
  <rightConditions>
    <concept type="TaskModule" model="sysc"/>
  </rightConditions>
  <actions>
    <copyPrimitive actionName="name" leftProperty="name" rightProperty="name"/>
    <transform actionName="thePorts" use="required" ruleName="Port-DPC">
      <leftproperty name="ports[*]"/>
      <rightproperty name="ports[*]"/>
    </transform>
    <transform actionName="theSignals" use="optional" ruleName="Connector-DPC">
      <leftproperty name="connector[*]"/>
      <rightproperty name="signals[*]"/>
    </transform>
    <transform actionName="theFeatures" use="optional" ruleName="Part-DPC">
      <leftproperty name="feature[*]"/>
      <rightproperty name="ownedInstances[*]"/>
    </transform>
  </actions>
</rule>
```

5 Conclusion

In our co-design environment, the transformation from UML to SystemC is a flow of successive transformations. In order to help in understanding our transformations, we have provided an example of transformation. The complete process from UML to SystemC simulation code is a set of three steps:

- From the application and the hardware architecture models to the association model
The mapping of the application onto the architecture is performed automatically by refactoring of a default mapping to satisfy some constraints expressed by the designer. It is an in-built transformation. This transformation aims at generating the association model according to the association metamodel.
- From the association model to SystemC simulation model
This transformation takes as input the association model generated at the previous level, the rules for transformations are expressed using `ModTransf`. The transformation engine generates the SystemC simulation model. Each concept in the input model is transformed to its corresponding concept in the SystemC simulation metamodel. This part was studied in this paper.
- From the SystemC simulation model to SystemC code
This last transformation is rather a code generation process. The same `ModTransf` tool is used too. The transformation here takes as input the previous model, some rules and

some code templates. These templates are called by the rules. They contains placeholders replaced by values of the concepts.

Model oriented co-design environment are widely used in the embedded system community. All the transformations between models could benefit of MDA techniques and **ModTransf** like tools. The reuse of models becomes a reality, add new models becomes feasible.

References

- [DBDM03] Cédric Dumoulin, Pierre Boulet, Jean-Luc Dekeyser, and Philippe Marquet. UML 2.0 structure diagram for intensive signal processing application specification. Research Report RR-4766, INRIA, March 2003.
- [Dum03] Cédric Dumoulin. MDA Transf: A model to model transformation engine, December 2003.
- [GK83] D. D. Gajski and R. Kuhn. Guest editor introduction: New VLSI-tools. *IEEE Computer*, 16(12):11–14, December 1983.
- [Obj00] Object Management Group, Inc. MOF meta object facility, specification, version 1.3. <http://www.omg.org/cgi-bin/doc?formal/00-04-03>, January 2000.
- [Obj03a] Object Management Group, Inc. MOF 2.0 query / views / transformations RFP. http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View_Transf._RFP.html, 2003. OMG paper.
- [Obj03b] Object Management Group, Inc., editor. (*UML 2.0*): *Superstructure Draft Adopted Specification*. <http://www.omg.org/cgi-bin/doc?ptc/03-07-06/>, July 2003.
- [pic03] picoChip. PC101 and PC102 datasheets. <http://www.picochip.com/technology/picoarray>, 2003.
- [TCKR03] T.Gardner, C.Griffin, A. Koehler, and R.Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. OMG document 03-08-02, July 2003. OMG document. Review of QVT proposals.