Mapping the MARTE UML profile to AADL

Skander Turki, Eric Senn and Dominique Blouin.

Labsticc Université de Bretagne-Sud Centre de recherche, BP 92116
56321 Lorient cedex, France
skanderturki@gmail.com, {eric.senn, dominique.blouin}@univ-ubs.fr

Abstract. CAT, the Consumption Analysis Toolbox, used with an AADL editor like OSATE allows for system-level power and energy consumption estimation. For MARTE users, such a tool is very valuable. This is why building a bridge from MARTE to AADL was essential. Transforming models is the solution that was adopted. This is why a MARTE to AADL mapping was needed. In this paper, we will present our mapping that is slightly different from that described by M. Faugere for crucial reasons.

Keyword. **AADL, MARTE, Mapping, Metamodeling, Model transformation.**

1 Introduction

AADL (Architecture and Analysis Design Language) is commonly used as a modeling language for real-time embedded systems [1, 2]. AADL models can be used by third-party tools to achieve an early analysis of the specification, the verification of functional and non functional properties of the system, and even code generation for the targeted hardware platform [3,4,5]. One of these third-party tools is CAT, the Consumption Analysis Toolbox, integrated with OSATE [**7**], an AADL editor.

In the other side, the MARTE (Modelling and Analysis of Real-Time Embedded Systems) profile [8] is commonly used for modelling and analysis of real-time embedded systems. As shown on the higher part of figure 1, it can permit, among other things, to depict the deployment of an application on a real-time operating system platform to build a Platform Independent Model (PIM) (independent of the hardware platform), and again, to depict the deployment of this PIM on a hardware platform building a Platform Specific Model (PSM). From there, the AADL tools environment might be used if an AADL PSM can be obtained from the MARTE PSM above. Two methods can be used, the first one is to create a UML AADL profile and to use it to annotate a MARTE PSM (see figure 1) then transform this UML/AADL PSM to an AADL model, the second method is to define a mapping of the elements of MARTE to those of AADL and to directly transform the MARTE PSM to an AADL PSM. This AADL PSM is then refined with platform specific information, for example a processor will be further refined by a technology specific reference (ARM7TDMI, PPC405, etc.).

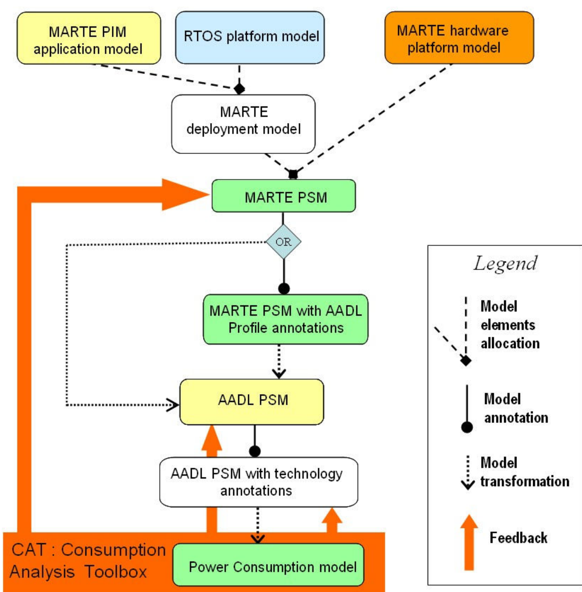

Fig. 1. Model-driven design: from MARTE to AADL.

In this paper, we describe the mapping of MARTE model elements to those of AADL focusing on the issues related to "modelling architectures"; issues related to the mapping of MARTE properties to AADL property sets are not treated. A conclusion will summarize the essential points of the paper and give some perspectives.

2 The *MarteToAadl* Model Transformation

We established a mapping between UML MARTE and the AADL language constructs that are slightly different from those described in the MARTE specification [8] (see Annex A: Guidance example of use of MARTE). The mapping we propose here is different for several reasons; the most important one is that we wanted our mapping to be an abstraction, which means that our transformation will abstract the MARTE model to an AADL model. The mapping presented by M. Faugere in [8] doesn't cover all the MARTE stereotypes. Then, it can only be used by MARTE users that already know that they are modelling for AADL as if they were using their UML tool as AADL editors where the MARTE profile is used partially and is used as an AADL profile. In our case, we target MARTE users that are modelling with MARTE without even knowing that AADL analysis tools could be used further. This is why we have to take into account all the MARTE profile, at least, when possible. The first case, where MARTE is used as an AADL editor is not useful as a specific AADL UML profile would be more appropriated.

The model transformation process we defined consists of an OCL-constraints-checking step [9] and a model transformation step which is the core transformation "*MarteToAadl*". The first step will verify that the input MARTE model respects some design rules to obtain a valid AADL model. Then the MarteToAadl model transformation is executed and the AADL model is produced.

3 The MARTE to AADL Mappings

A. The packages hierarchy issue

An AADL model has a flat hierarchy (figure 2). This means that it has only one root element that represents the AADL specification in which there can be leaf packages (Packages do not contain other packages). In UML, a root model contains packages that contain other packages; it is a tree-like hierarchy. In AADL packages hierarchy can be simulated through file system directories. An AADL model has this hierarchy:

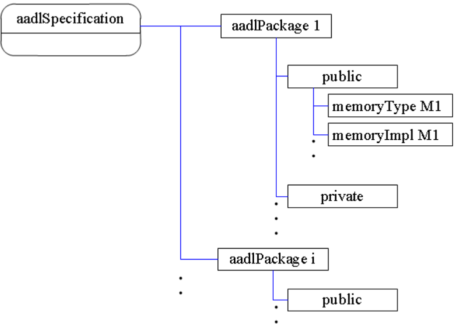

Fig. 2. AADL Model Hierarchy.

In UML, the hierarchy is also a design information that has its meaning to the designer understanding of the system. This is why, when we transform the MARTE model to AADL we need to keep track of this information. We use namespaces for this: " *XUPWithMJPEG::PK_Application::Package_IPC* "

This gives us the following ATL transformation rule for packages:

```
  rule Packages {
from s : UML2!Package (not s.oclIsTypeOf(UML2!Model))
to p: AAXL!AadlPackage
(name <- s.qualifiedName,       …)}
```

Here, *qualifiedName* of the UML packages returns the name with the namespace.

B. Mapping UML Class diagram hierarchy to AADL Category Types and implementations hierarchy

In a first approach we used the *isAbstract* Boolean field of UML classes to differentiate between an AADL type and an AADL implementation; this was a solution to avoid using additional stereotypes. But we finally decided to create an AADL type and an AADL implementation for each UML class because the first approach was like using MARTE as an AADL editor. With the first approach we had to add many OCL constraints to verify we did not have semantically illegal AADL configurations. For example, an abstract class that inherits from a non-abstract one would lead to an AADL type extending an AADL implementation which is illegal in AADL. We also had to choose which UML relationships would be mapped to the AADL implementation relationship and which one would be mapped to the AADL extension relationship. We distinguished three cases of UML configurations (figure 3) that can be seen as an AADL equivalent to:

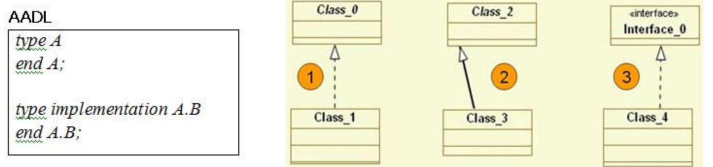

Fig. 3. Possible AADL type implementations with UML (Name with Italics = isAbstract)

We noticed that we couldn't use an interface to depict an AADL Type because a UML interface is not an *encapsulatedClassifier* (cannot have ports). In fact, we can only use classes to define ports, which is an important aspect in architecture modelling.

We could use the realization to depict an AADL type but in the composite diagram, in an implementation element we would not be able to inherit the ports declared in the corresponding class, we could only do that with the inheritance, in addition to this, we thought we would reserve this notation to the bus/data access AADL construct. This is why only the second case (figure 3) was possible.

We also had to prevent someone from using a generalization between two classes that do not correspond to the same category (processor, bus, thread…). This was the role of the pre-transformation verification step which induced more OCL constraints.

We also mapped the use of a generalization between two abstract classes from the same component category or between two non-abstract classes from the same component category to the extension mechanism in AADL.

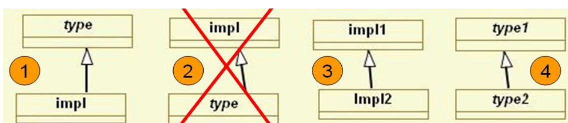

Fig. 4. Four combinations of generalizations could be found in UML.

Four possibilities can be drawn using generalizations in UML (figure 4). Each possibility corresponded to a different AADL construct:

- The first one mapped to the AADL implementation
- The second becomes meaningless in AADL.
- The third and fourth situations mapped to the AADL extension construct.

This first choice led us to a complicated mapping that, as we already said, was using MARTE as an AADL editor. It also generated too much OCL constraints that impacted dramatically the designing activity as the designer had to know all these imposed rules in addition to the design language. We finally chose a different mapping that is much easier for the designer, who doesn't need to know more design rules than those of UML and MARTE. This mapping consists on splitting each class into a type and its implementation and to map the UML generalization to the "extends" mechanism of AADL as shown in the following table 1 (here the "extends" AADL mechanism is depicted with the black-headed arrow with discontinuous arc, the blue-headed arrow in AADL depicts the "implements" AADL mechanism).

The following example (figure 5) illustrates how this mapping can translate any UML/MARTE configuration to AADL with only one assumption on the initial MARTE model.

TABLE 1
MAPPING UML CLASSES AND GENERALIZATIONS TO AADL

Num	UML configuration	Equivalent AADL configuration	Description
1	Class	Type Impl	Each non-abstract UML class is transformed in AADL to a Type that defines its interface and an implementation that defines its composition.
2	*Class*	Type	Each abstract UML Class is transformed to a type with no implementation.
3	ClassA ClassB	TypeB TypeA ImplB ImplA	In this case : ClassB and ClassA are non-abstract classes, so each one is transformaed to a Type and its implementation. The generalisation relationship will be transformed to an extension between TypeB to TypeA.
4	ClassA *ClassB*	TypeB TypeA ImplA	Same thing as situation 3, but TypeB has no implementation because ClassB is abstract.
5	*ClassA* ClassB	TypeB TypeA ImplB	Opposite situation as in 4.

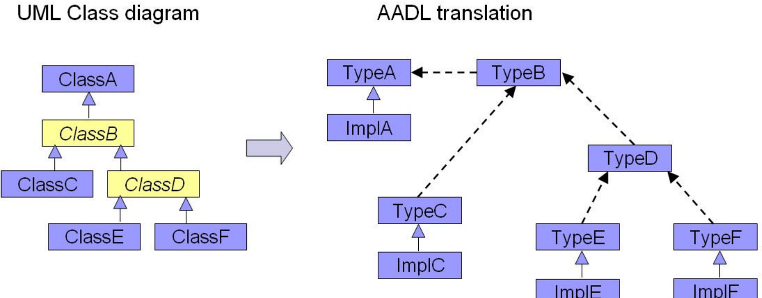

Fig. 5. UML class diagram translation. (*ClassB* and *ClassD* are abstract)

With this mapping we will never have extensions between implementations, which is not a problem as long as both representations are equivalent. This mapping also guarantees that each UML generalization's hierarchy can be transformed to an equivalent extension's hierarchy in AADL.

C. Mapping MARTE constructs to AADL component categories

MARTE offers more detailed design possibilities than AADL. In fact, all AADL constructs have their mapping inside MARTE but the other way is not true. For example, in MARTE we have a *HwProcessor* and a *HwComputingResource*. Both of them can be translated in an AADL processor but the design detail in AADL will be of coarser granularity. Another example is the *HwBridge* in MARTE that has no equivalent in AADL. It must therefore be abstracted in AADL to a general *device* that corresponds in MARTE to *HwDevice*.

As we already said, we tried to make this mapping an abstraction of MARTE to AADL. This is why we mapped each AADL component category to a set of MARTE model elements.

TABLE 2
MAPPINGS ESTABLISHED IN THIS SECTION

MARTE Construct	*AADL Construct*
'HwProcessor', 'HwASIC', 'HwPLD', 'HwComputingResource', 'ComputingResource', 'ProcessingResource'	processor
'DeviceResource','HwActuator', 'HwArbiter', 'HwBridge', 'HwDevice','HwDMA', 'HwCLock', 'HwCoolingSupply', 'HwComponent', 'HwI_O', 'HwISA', 'HwMedia', 'HwMMU', 'HwPowerSupply', 'HwResource', 'HwSensor', 'HwStorageManager', 'HwSupport', 'HwTimer', 'HwTimingResource', 'HwWatchDog'	device
'HwBus'	bus
No stereotype, SysML::Block	system
'StorageResource', 'HwMemory','HwCache','HwRAM','HwROM', 'HwDrive'	memory
'MemoryPartition', 'RtUnit', 'DeviceBroker', 'MemoryBroker', 'SynchronizationResource', 'MutualExclusionResource', 'SwMutualExclusionResource', 'SwTimerResource'	process
'SwSchedulableResource', 'SchedulableResource', 'ConcurrencyResource'	thread
none	threadGroup
'Type', 'PpUnit', 'Alarm', 'InterruptResource', 'MessageComResource', 'SwCommunicationResource', 'NotificationResource', 'SharedDataComResource', UML2!DataType or UML2!Signal	data
An operation (class method) in a thread or a subProgram	subProgram

Ideally, the AADL *system* component category should be described by a stereotype like the *SysML::Block* but we did not want to impose a dependency on another UML profile for only one stereotype. This is why we also map a class that do not have a stereotype to the general concept of system.

Some stereotypes in MARTE cannot be abstracted to an AADL construct:

- Because it is too abstract: for example 'TimingResource' can either be a hardware or software component. For such cases we chose a default mapping (see table 3 rows 1 and 2) so that to have a complete mapping. But, using an OCL constraint of low severity, we generate a warning to the user to let him know of the issue.
- Because this aspect of the design is not supported at all by AADL. For example, the 'HwEndPoint' or the 'HwPort', these aspects are not depicted in AADL. In such cases, we simply don't consider the information but OCL warning constraints will inform the designer about these issues (see table 3 row number 3)

TABLE3
OCL WARNING CONSTRAINTS FOR NON ABSTRACTED MARTE ELEMENTS

Defined ocl method	MARTE Construct
tooAbstDefMappedToDevice()	'TimingResource', 'TimerResource', 'HwCommunicationResource', 'Resource', 'ClockResource'
tooAbstDefMappedToBus()	'CommunicationMedia'
unsupportedMarteStereotype()	'HwPort', 'HwEndPoint', 'ResourceUsage', 'Scheduler', 'Clock', 'ClockType'

TABLE 4
COMPOSITION POSSIBILITIES BETWEEN AADL COMPONENT CATEGORIES

AADL Construct	Possible Sub-Components
processor	memory
device	none
bus	none
system	System, data, process, processor, memory, bus, device
memory	memory
process	Thread, data, threadGroup
thread	data
threadGroup	data, thread, threadGroup ***(not mapped to MARTE)***
data	data
subProgram	None ***(not mapped to a classifier in MARTE, composition is irrelevant)***

In AADL the *SubProgram* category is a "sequentially executable source text". This corresponds in UML to an Operation in a Class. In AADL a *SubProgram* Type can have different implementations. In UML an operation has methods that can be an *OpaqueBehavior* (natural language, programming language…), a state machine or an activity. We can then use *OpaqueBehaviors* as methods of an Operation to depict different AADL *implemententions* of an AADL *SubProgram Type*. This is the more natural mapping we could find with UML. The difference is that a *UML::OpaqueBehavior* is contained in the class where it is defined while in AADL a *SubProgram* is defined independently and instantiated inside another component. But as we are transforming a more constrained language (in this aspect) to a more general language, this will not lead to different semantics.

D. Mapping UML Class properties to AADL constructs

UML Class properties that are typed by a class that is mapped to an AADL component category are transformed to subcomponents typed by the corresponding AADL implementation components.

UML properties that do not correspond to an AADL component category are transformed to AADL properties and a new AADL property definition is created for each one.

UML properties of abstract classes must be propagated to all specializing classes. This is simply done by using the UML API getAllAttributes() method of a *UML::Classifier* instead of the more common Classifier.attribute property:

```
memorySubcomponent <- s.getAllAttributes()
->select(z | z.attIsCompCateg (s.memoryMapping()))
    ->collect(e | thisModule.myMemorySubComponent(e)).flatten()
```

Composition in AADL is restricted by some contraints (see table 4). These constraints are applied in our mapping using the *select* OCL filter. In the example above, the memorySubComponent AADL component attribute will only contain elements that are memories using the select statement:

```
->select(z | z.attIsCompCateg (s.memoryMapping()))
```

A result of this is that illegal properties typing in UML/MARTE will be ignored. For example if a UML class stereotyped "MemoryPartition" (mapped to AADL process) had a property typed by another Class stereotyped "HwBus" (mapped to AADL bus), no error will be raised as the select statement will just ignore the illegal properties. This is why OCL warning constraints are added to the OCL verification step. The following OCL code shows an example of a warning constraint whose result will be used to tell the designer that a Class property will be ignored:

```
-- 0 -- Process Impl sub-components can only be threads or data
context Class def : processContainsOnlyThreadOrData() : Boolean
=  (self.isProcess()) implies   (self.attribute
          ->select(p | p.type.oclIsTypeOf(Class))
          ->select(s | s.type.isACategoryType())
          ->forAll(a | a.type.isThread() or a.type.isData())  )
```

E. Mapping UML interfaces usage and realisation

The UML/MARTE designer defines types for ports. Two types of ports must be defined:

1. Types for ports that will connect to components.
2. Types for ports that will be provided by a component to allow other components to be connected to it.

Then, the designer sets these types (that are also classes) for the ports of the classes. Finally, he connects the corresponding ports. This configuration is mapped to the "requires bus access" AADL construct (same for data, see figure 6).

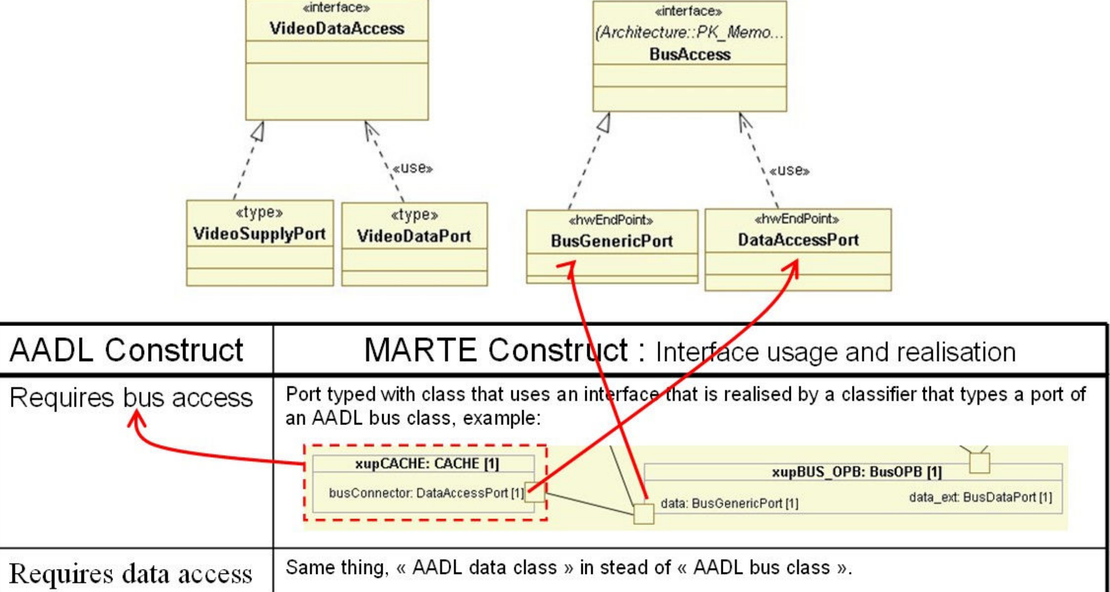

Fig. 6. UML interface usage and realisation mapping to AADL constructs.

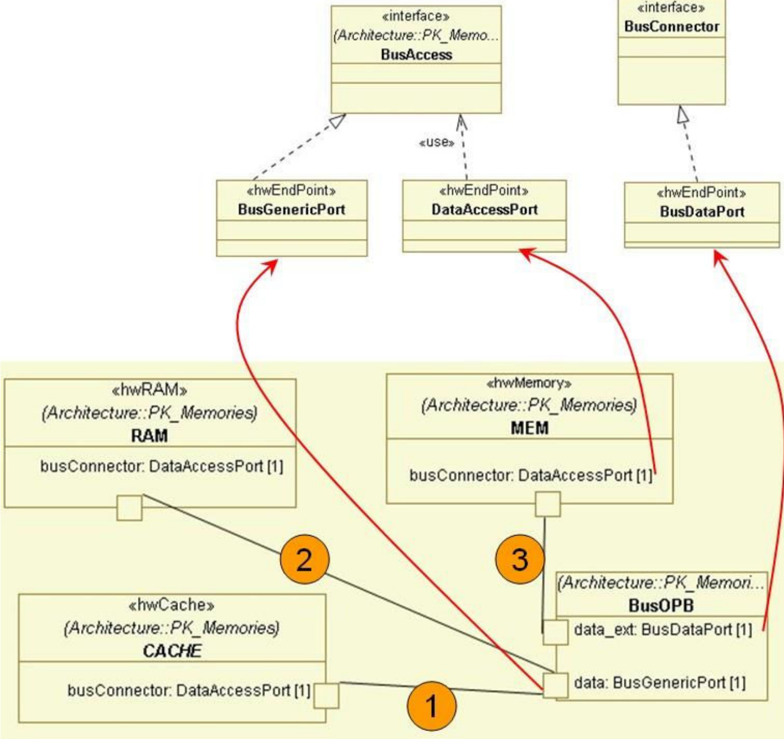

Fig. 7. Design examples mapped to the "requires bus access" configuration.

This configuration (figure 7) is mapped to the "requires bus access" features for a system, memory, processor, bus and device categories: In this example, each element that has a port typed with the *DataAccessPort* class will have a « required bus access » element to any bus that has a port that realizes the *BusAccess* interface. See in this example (figure 7) the connections 1

and 2 where both *busConnector* ports are typed by *DataAccessPort* which uses the *BusAccess* interface that is realized by the BusGenericPort class that types the data port in the *BusOPB* class.

In UML, one can connect two ports even if the interface used and realized is not the same. See in this example the connection 3 where the *busConnector* port is connected to a port that is typed by a class that realizes another interface than *BusAccess*. This situation should cause at least a warning. It is a result of the informality of UML. In fact, in UML, only "compatible" ports can be connected [10]. This means that we have to define what « compatibility » between ports means. This is a UML « *semantic variation point* ». Computer science doesn't like semantic variation points; this is why we had to define this "compatibility". We define it this way*: Connectable elements attached to a connector are compatible if they are typed by a class that either uses or realizes the same interface. At least one must use the interface and one must realize it.*

F. Ports connected to their owning Class properties in UML/MARTE

A class property exposed through data or bus typed port can be translated to « **Provides data/bus access** ». Two situations can be found:

- Situation 1: The property is directly connected to the boundary port.
- Situation 2: The property is connected to the boundary port through one of its own ports.

Both situations are supported by our mapping and the data/bus provided classifier will be the one that is directly connected to the boundary port.

Two examples are shown here in figure 8:

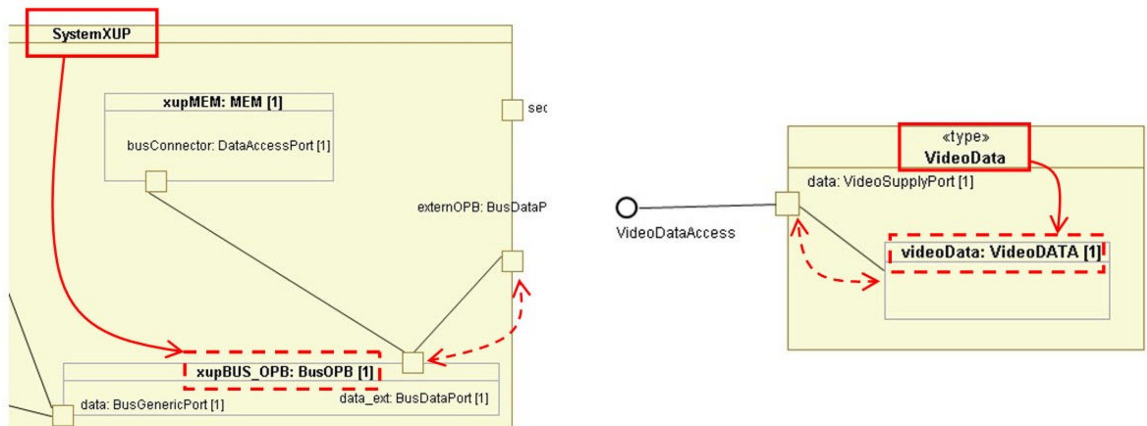

Fig. 8. Examples of the two situations of the port connected to its own Class properties.

TABLE 5
PORTS CONNECTED TO THEIR OWN CLASS PROPERTIES MAPPINGS.

AADL Construct	MARTE Construct : Interface usage and realisation
Provides bus access	A class that has an AADL bus typed property connected to a boundary port, either through a port (left figure) or directly (right figure), is tranlated in AADL as providing bus access.
Provides data access	Same thing, « AADL data typed property » in stead of « AADL bus typed property ».

G. Mapping UML/MARTE ports to AADL constructs

M. Faugere [8] has defined a mapping of UML/MARTE ports to AADL. Our mappings are mappings of MARTE to AADL so it has to cover more possibilities to abstract more situations to AADL. In fact, M. Faugere's mapping is a mapping of AADL to MARTE, i.e, it answers the question: "how an AADL design can be specified using MARTE?". We, are answering a different question: "How can we translate a MARTE specification to AADL?".

The AADL data port corresponds in UML to a standard *UML::port*, as “Ports represent interaction points between a classifier and its environment” [10]. The MARTE *flowPort* adds, among other things, the direction. In fact, in MARTE, “FlowPorts have been introduced to enable flow-oriented communication paradigm between components” [8]. An AADL event/(event data) port corresponds to a message port or a ClientServerPort. In [8], “Message ports support a request/reply communication paradigm”. Finally, the difference between an event port and an event data port in AADL is that an event data port is typed (has a *dataClassifier* property in the meta-model).

TABLE 6
UML/MARTE MAPPINGS

MARTE	AADL	Direction
flowPort	data port	In : in / Out : out / inOut : in out
1 – messagePort or clientServerPort (not typed or typed by a UML signal that do not have data attributes) Or 2 - UML standard port (not typed or typed by a UML signal that do not have data attributes)	event port (in AADL it doesn't have a classifier)	*1 - out : messagePort direction = required* *in : messagePort direction = provided* *2 - inout : default*
1 – messagePort or clientServerPort typed by data (except signals that do not have data attributes) 2 - UML standardPort data typed (except signals that do not have data attributes)	event data port (In AADL it has a data classifier)	*1 - required : messagePort direction = required* *provided : messagePort direction = provided* *2 - required : none* *provided : none*

4 Conclusions and Perspectives

The mappings described in this paper are general and can be used without letting the designer know about the details of our mappings, except for the description of the ports and their typing and for the semantics that are imposed by the AADL language.

The syntactic differences between the source and the target language are well handled by the transformation rules as the ATL tool we used make it possible to call Java code through which we can do everything a computer can do. Other languages that are only based on declarative rules may provide an incomplete transformation. It is therefore important to have both declarative rules and imperative programming. The declarative rules simplify the definition of the rules when the mapping is simple enough. But in some cases the use of the imperative programming is indispensable.

The central question that is raised by model transformations is the semantics gap issue. The question “how can we solve semantics gap?” needs to be answered. Reading the definitions of a modelling aspects in both source and target specifications and then comparing them can become really heavy work. Some mappings designed for M2M transformations are built without going into that level of detail and that inevitably gives an apparently acceptable translation but the underlying semantics of source and target models would have different meanings.

Resolving a M2M transformation is a recurrent problem that needs to be undertaken with more systematic approaches to become more efficient. This systematisation question can be answered by describing "M2M resolution patterns". Some patterns have already been adopted but not precisely described like the decomposition of the transformation unto a chain of n transformations where each one is a simple transformation between two semantically close meta-models. Describing these patterns will take model transformations from the state of a handwork process to that of an engineering one. This paper is not an M2M resolution patterns paper, but we present here some other evident patterns we have encountered:

- The abstraction pattern: A source model element can be mapped to a target model element that is more general, that has less detail like abstracting a HwSensor to a general Device.
- The projection pattern: The target domain is less expressive than the source one, then a projection is done to reduce the "dimensions" of information into the target domain. Like transforming a model element from a language with static and dynamic descriptions to a pure architectural language.
- The design-banning pattern: The source domain is less rigorous is term of modelling possibilities; some cases are non-sense if transformed to the target domain as is. Banning these cases using a constraints verification phase is a solution to this problem.

References

1. P. Feiler, B. Lewis, and S. Vestal, The sae architecture analysis & design language (AADL) A standard for engineering performance critical systems, in IEEE International Symposium on Computer-Aided Control Systems Design, Munich, October 2006, pp. 1206–1211.
2. SAE - Society of Automative Engineers, SAES AS5506, v1.0, Embedded Computing Systems Committee, SAE, November 2004.
3. T. Vergnaud, Modélisation des systémes temps-réel embarqués pour la génération automatique d’applications formellement vérifiées, Ph.D. dissertation, Ecole Nationale Supérieure des Télécommunications de Paris, France, 2006.
4. A. Rugina, K. Kanoun, and M. Kaniche, Aadl-based dependability modelling, LAAS, Tech. Rep., 2006, number = 06209.

5. J. Hugues, B. Zalila, and L. Pautet, Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina, in Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07). IEEE Computer Society Press, may 2007, pp. 106–112, porto Alegre, Brazil.
6. The SPICES ITEA Project Website. [Online]. Available: http://www.spices-itea.org/
7. OSATE homepage, http://www.aadl.info/aadl/currentsite/tool/osate-down.html.
8. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems Beta 2, OMG Adopted Specification, OMG Document Number: ptc/2008-06-09, http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf.
9. S. Turki et Al, Checking syntactic constraints on models using ATL model transformations, Workshop mtATL2009 Nantes, 08-09 july 2009.
10. UML 2.2 OMG official specification, UML superstructure specification, 2009/02/02, www.omg.org/spec/UML/2.2/Superstructure/PDF/