Component-based Models Going Generic : the MARTE Case-Study

César Olavo de Moura Filho, Anne Etien, Julien Taillard, Cedric Dumoulin, Frédéric Guyomarc'H

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Component-based Models Going Generic: the MARTE Case-Study

César Olavo de Moura Filho — Anne Etien — Julien Taillard — Cédric Dumoulin — Frédéric Guyomarc'h

N° 6632

September 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6632--FR+ENG

Component-based Models Going Generic: the MARTE Case-Study

César Olavo de Moura Filho* , Anne Etien† , Julien Taillard‡ , Cédric Dumoulin§ , Frédéric Guyomarc'h¶

Thème COM — Systèmes communicants
Équipes-Projets DaRT

Rapport de recherche n° 6632 — September 2008 — 15 pages

Abstract: One of the reasons for using component-based modeling is to improve on reusability. However, there are cases where a whole component cannot be reused just because one element from its internal structure does not present the required features (e.g., type, multiplicity, etc). In this paper, we propose the use of parameterized components as a way to address this problem - and thus to get a further boost on reusability. The UML specification provides support to parameterization via templates. However, when it comes to component-based modeling, UML is but the first metamodel in sometimes long chains of transformations, comprising other domain metamodels. So, in order to keep parameters deeper down the transformation chains, we introduce generic components in those metamodels. However, instead of changing the target metamodel, we decided to create an independent metamodel with the additional concepts required by parameterization, so it can be attached to any target metamodel. The most obvious advantage of this approach is that we do not have to touch the target metamodel. We also demonstrate how existing transformations can be easily adapted to accept the parameter-related concepts. To illustrate our ideas, we used OMG's MARTE metamodel for real-time and embedded systems. The approach has been validated through transformations written in QVT.

Key-words: Model Driven Engineering, Component-based Modeling, Model Transformation, QVT, UML, MARTE standard profile

* *cesar.de_moura@inria.fr*
† *anne.etien@lifl.fr*
‡ *julien.taillard@lifl.fr*
§ *cedric.dumoulin@lifl.fr*
¶ *frederic.guyomarch@lifl.fr*

Centre de recherche INRIA Futurs
Parc Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 Orsay Cedex
Téléphone : +33 1 72 92 59 00

Vers la généricité des modèles orientés composants : le cas d'étude MARTE

Résumé : L'un des avantages principal de la modélisation orientée composant est d'améliorer la réutilisabilité. Cependant, il existe des cas où un composant ne peut pas être réutilisé dans sa globalité uniquement parce qu'un élément de sa structure interne ne satisfait pas les caractéristiques attendues (par exemple le type, la multiplicité, etc.) Dans ce rapport, nous proposons d'utiliser des composants paramétrés afin de résoudre ce problème - et donc d'améliorer la réutilisabilité. La spécification UML propose le mécanisme de *template* comme support à la paramétrisation. Cependant, UML n'est parfois que le premier métamodèle d'une longue chaîne de transformations comprenant d'autre métamodèles de domaine. Ainsi, afin de garder les paramètres plus profondément dans la chaîne de transformation, nous introduisons des composants génériques dans ces métamodèles. Cependant, au lieu de changer le métamodèle cible, nous avons créé un métamodèle indépendant avec les concepts nécessaires à la paramétrisation afin qu'il puisse être attaché à n'importe quel métamodèle. Le principal avantage de cette approche réside dans le fait qu'il n'est pas nécessaire de modifier le métamodèle cible. Nous démontrons aussi comment les transformations existantes peuvent être facilement adaptées pour supporter les concepts liés à la paramétrisation. Pour illustrer ces idées, nous avons utilisé le métamodèle MARTE de l'OMG pour les systèmes temps réel embarqué. L'approche a été validée par des transformations écrite en QVT.

Mots-clés : ingénierie dirige par les modéles, profil standard MARTE, paramètres, Modélisation orientée composant, Généricités,

1 Introduction

Model Driven Engineering (MDE) proposes doing away with contemplative models by turning them into productive artifacts sitting right at the beginning of a process that finishes with executable code. Being models more abstract than programming code, changes can be more easily operated on them, while model transformations assure that such changes will be rippled to the final code, thus incurring in lower development time and costs.

Component-based modeling can also benefit from the MDE main rationale. However, in practice, we regularly come across situations in which component-based models are not abstract enough so as to be reused in different contexts. Sometimes a previously designed component just cannot be reused as is, even if most of its specification corresponds to the designer's needs. For example, one may rule out an otherwise perfectly suitable component just because the multiplicity of one of its ports is not adequate. In this case, "reuse" boils down, in practice, to making a copy of the component and to manually performing the required changes. This procedure, in addition to being error-prone, incurs in extra work - a work that risks to be repeated anytime a given component cannot be reused exactly as is.

A solution to this problem can be the use of parameterizable components, that is, components whose specification is incomplete, comprising placeholders that can be later replaced by actual model elements. More specifically, parameterizable components are not fully specified in the moment they are specified, leaving some "blanks" (i.e., the parameters) to be filled out later on, once we know their actual values. Parameterization has been widely used in different programing languages (like C++ templates and Java generics [**?**]), and in modeling languages, like EMF generics [**?**] and UML templates [**?**] - this one, of special interest for this report.

With UML templates, parameterization can be introduced from the outset of the modeling process. However, the genericity achieved with the introduction of templates is eventually lost once parameters get their values - something done by the model designer, much too early in the development process. Considering that UML models are sometimes used in chains of transformation as the primary artifact, out of which other models - and eventually programming code - are derived, we would like to be able to continue with generic components in the derived models as well – as opposed to resolving parameters right after we leave UML.

So, if we somehow manage to push the assignment of values further down a transformation chain[1], parameters could alternatively have their values programmatically assigned by optimization/refactoring algorithms, called from inner transformations. We also reap some benefits of bringing parameters up to the generated code, since this would allow us to produce generic code - which can be distributed as a library and reused many times over, without ever having to revisit the transformation chain.

A way of achieving this goal is by enriching domain metamodels with support to parameterization and having parameters propagate through the successive metamodels until they reach the one that will effectively make use of them -

[1] Transformation chain corresponds to successive transformations in which the target metamodel of one transformation is the source of the following one.

ultimately up to the generated code. In this report, we propose a solution to the parameterization of metamodels. Although this solution can be general enough to fit any metamodel, we explore here only component-based metamodels.

The rest of this report is organized as follows. In section 2, we describe the UML solution for parameterization. In section 3, we discuss an example that illustrates our needs in terms of parameterization. Section 4 presents our solution to parameterize existing metamodels. Section 5 explains how to apply our approach to the OMG MARTE metamodel. The final section draws the conclusions and suggests future work.

2 Parameterizing UML Models

This section describes the UML solution to parameterization. Additionally, it raises some issues relative to its usage and proposes solutions to them.

UML concepts. Parameterization has been introduced in the UML specification through the use of *Templates*. The UML templating mechanism is based on two main elements: the *TemplateableElement* and the *ParameterableElement*. The former, referred to simply as *template*, is the 'generic' element, which has parameters and can have its inner parts *exposed* (to use UML jargon) by the parameters. UML elements that can play the role of *TemplateableElement* are classifiers, packages, operations and even string expressions. In turn, *ParameterableElement* represents the template's inner elements than can be exposed -i.e. the ones which will be replaced by the actual values. Classifiers, features (properties and operations) and value specifications are some UML elements that can act as *ParameterableElement*.

(a) A templated class

(b) The binding operation

Figure 1: Declaration and binding of a parameter in UML

Figure 1.a illustrates a templated class (*List*). Graphically, a templated class is expressed through the small dashed rectangle superimposed to the symbol of the class, while its parameters are listed inside the rectangle ('E', in this specific case). Once defined, templates can be used by other elements of the model. This is achieved by means of an operation called *binding*, in which actual value(s) are substituted for the parameter(s) (Figure 1.b). A binding is expressed through the *templateBinding* relationship, a directed relationship going from the bound element to the template and labeled with the «bind» stereotype. In our example, *ListOfString* is obtained from *List* by replacing E (in reality, by replacing the ParameterableElement pointed at by E - not shown in the figure) by a String class.

Issues related to UML templates and their implementation in UML tools. In our current practice, a few situations have been detected

that cannot be properly modeled with the UML template solution, raising some issues, due to both a faulty UML specification and to incomplete UML tools.

One of these issues concerns the fact that the UML specification does not allow one given parameter to expose more than one parameterable element. Unfortunately, this happens to be a very harsh constraint that prevents us from modeling some useful situations frequently found in our practice. For example, the simple class illustrated in Figure 2 cannot be properly specified with the UML templating mechanism. In this example, class A has two parameters, T (of type *Class*) and n (*Integer*) and two properties ($prop1$ and $prop2$), which are both arrays of type T and size n. While this configuration works well with the parameter T (i.e. both properties can have the same type: the class that will be bound to T) it fails with the parameter n (i.e. two multiplicities cannot be specified by the same parameter). A quick analysis of the UML specification shows us that the references between a property and its type and between a property and its upper and lower bounds present different natures, the first being a simple reference and the second, an aggregation. This issue has been reported to the OMG and, hopefully, it will be corrected in the next versions of the specification. In the meantime, we have to make do with a home-brewed solution - though further explanation is beyond the scope of this report.

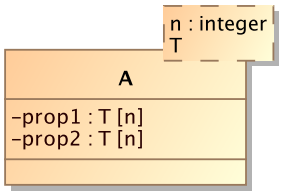

Figure 2: A parameterized class that can not be designed

Excepting this issue - the only serious threat to our modeling efforts - the UML template specification provides all the mechanisms necessary to express genericity. Some of them are, however, too heavyweight and clumsy to be used for our purposes. An example of such clumsiness - and this constitutes the second type of problem, hinted at above - is the graphical representation of the binding, with its additional box and arrow, which tends to clutter even more the usually crammed up component diagrams. We decided to address this problem by making use of the *anonymous binding* feature. Barely mentioned in the UML specification, the *anonymous binding* is used to encode the parameter substitution information directly in the bound class name, dispensing with the explicit binding. With anonymous binding, the ListOfString class of Figure 1 would be obtained just by writing[2]:

`List<T→String>` .

This handy notation will certainly spare us some precious space from our component diagrams. In the next section we will provide an example model that uses both UML templates and component-based elements.

[2] Due to the fact that anonymous binding is completely ignored by existing tools, we had to perform a slightly adaption in the official UML notation: `myListOfString<T→String>: List`

3 Designing a Component-based Model with Parameters

In this section, we describe an application for scientific computing as an example to clarify our component model. Since a full-fledged parallel solver is far beyond the scope of this article, we provide just a streamlined example, but that includes all the concepts we need. All these concepts are described in details in the MARTE profile for UML [?], though a some of them are further detailed in the Section 5 of this report.

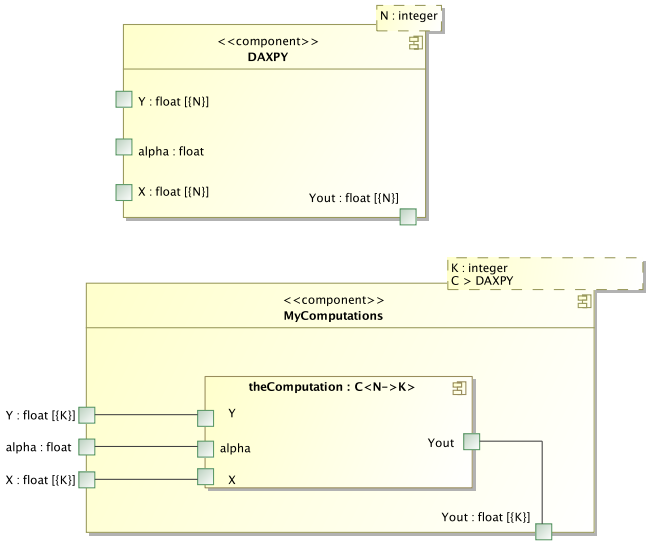

Figure 3: Definition of a parameterized component with UML templates

The first concept used in our models is the component itself. Here, in figure 3, the DAXPY component represents vector computations ([?]). Data are exchanged through flow ports stereotyped as *in* or *out*. Here the inputs are two vectors X, Y and a scalar *alpha*; the output is the linear combination $alpha * X + Y$. All these ports have a shape which represents the size of the data going through it, and this shape can be multidimensional. To avoid re-design the model whenever we need to change the shape, a parameter ($N : Integer$) was introduced. This component can then be used to perform actual computation: here it appears as a part of *MyComputations* component, which can be regarded as the main program. This component is also parameterized with the size of the data chunk it can deal with per operation. We also wanted to be able to change the actual implementation of the DAXPY component itself. One can think of this example as if we had a very optimized function but which can only run on a few processors (like, for example, the Intel Math Kernel Library [?],optimized for the Intel processors). If we wanted to run the program on another architecture, a different implementation would be needed. Figure 4 illustrates this situation.

Figure 4 depicts two components that implement DAXPY (*NetLib_Reference_DAXPY* and *MKL_DAXPY*), and they are used by two different programs. *Program1* uses the non optimized function *NetLib_Reference_DAXPY* with data size of 5 (i.e. K = 5) , whereas

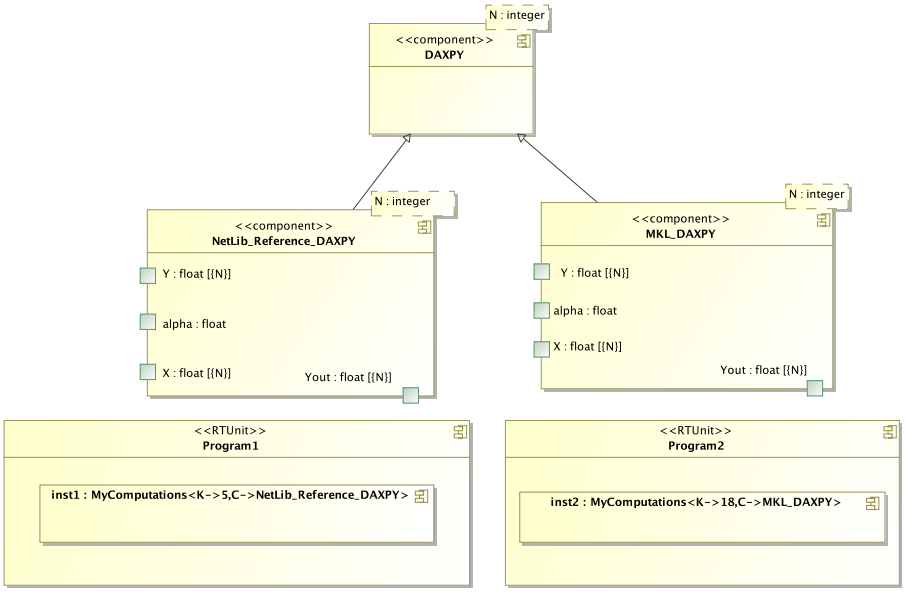

Figure 4: Instanciation of the parameterized component

Program2 uses the Intel optimized function *MKL_DAXPY* with bigger data chunks (K = 18). Likewise, the type of the C parameter is assigned to *NetLib_Reference_DAXPY* and *MKL_DAXPY*, respectively.

Despite its simplicity, this example represents well the kind of genericity we need in our models. In the next section, we present a way of extending the support to parameters to beyond UML, so we can preserve the genericity illustrated in this example in the derived models.

4 Parameterizing component-based metamodels

4.1 Overview

In this section, we describe our solution for adding parameterization capabilities to a component-based metamodel. Since the metamodel is expressed in Ecore, the first solution that crosses our minds is to make use of the EMF support of generics. However, this hasty conclusion turns out to be a bad choice, since, like Java generics, EMF generics is a purely type-based parameterization mechanism. That is, contrary to UML, parameters can only expose types in typed elements (something that roughly corresponds to UML's ClassifierTemplateParameter), whereas one of our main interests in using parameterization is to expose bounds of multiplicity elements (i.e. upperValue and lowerValue). So, we promptly ruled out any solution based on the EMF generics. Another point to be considered is that, since there already exist much ongoing work based on the MARTE metamodel, the proposed solution could not be too disruptive, in particular to the existing transformations.

Recapitulating what has been discussed so far in terms of requirements for our parameterization needs, we look for a solution that:

- does not overload diagrams with parameter-related notation
- allows parameters to expose more than one element
- is independent of any particular metamodel

To meet the first two items, we had to overcome the problems pointed at in the previous sections, namely the impossibility of a same parameter to expose more than one parameterable element and the limitation of existing UML tools, which do not implement the anonymous binding feature. Granted, these are temporary solutions waiting for the enhancement of UML specification and tools.

In addition to the MARTE metamodel, we use in our research work other more specific metamodels for embedded systems. It is important, then, that our solution for adding parameterization capabilities to MARTE can be reapplied to the other metamodels - located downstream the transformation chain - without too much effort. So, in order to meet the last requirement above, instead of modifying every metamodel we use so as to accommodate the additional concepts related to parameterization, the adopted approach was to create a new, completely independent metamodel with such concepts, but that sits on top of the target metamodel. This metamodel has to be generic enough, making no assumption about the target metamodel it will hang up to. Nevertheless, for the example described in this report, we will - somewhat artificially - constrain it to component-based models.

We will now give a glimpse of the proposed metamodel. For the sake of clarity, we will break it down into two parts: the first part, concerning the specification of the generic component and the second part, regrouping the concepts related to the binding process.

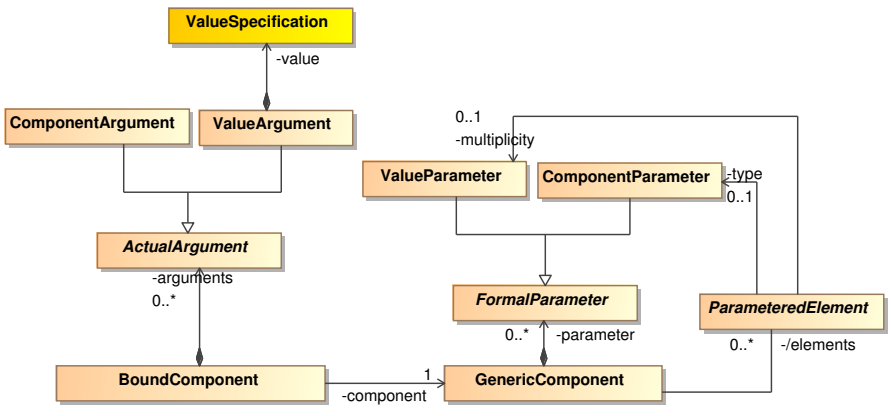

Figure 5: Parameter Metamodel

4.2 Generic Metamodel Description

The right part of Figure 5 highlights the concepts involved in the definition of a generic component. It comprises the central element, *GenericComponent*,

which is the element that will be parameterized. In turn, *GenericComponent* contains one or more *FormalParameters* and one or more *ParameterizedElements*. These three elements are directly obtained from the corresponding UML elements, respectively *TemplateableElement*, *TemplateParameter* and *ParameterableElement*. Moreover, a *FormalParameter* is further subdivided into *ValueParameter* and *ComponentParameter*, thus reflecting our need of exposing, respectively, values (e.g. the multiplicity of a port from a component) and types (e.g. the type of a component's part) from the component's internal structure.

The left part of the metamodel refers to the binding process: it is constituted of a *BoundComponent*, which may contain one or more arguments (*ActualArgument*). Mirroring *FormalParameter*, *ActualArgument* is further subclassed into *ValueArgument* and *ComponentArgument*, thus taking account of both types of elements that can replace parameters. *BoundComponent* is the element that represents a *GenericComponent* whose parameters have been given values. Contrary to the other elements described so far, it does not have a direct counterpart in the UML templating model, and, consequently, cannot be generated directly from a UML concept.

4.3 Assigning values to parameters

The crucial part of the binding process is the assignment of values to the formal parameters. Before it, we have two would-be components, one, *GenericComponent*, containing the structure of a full-fledged component (i.e., ports, parts, connectors, etc.), but lacking some values for a complete specification; and the other, the *BoundComponent*, containing nothing but the needed values (referred to by the *ActualArgument*s). So, in this step, a brand new *StructuredComponent* is created that has the structure of the *GenericComponent* but that replaces all *FormalParameter*s by the value referenced by the matching *ActualArgument*.

In the next section we apply the metamodel here described to the OMG MARTE metamodel, so as to make it parameter-aware.

5 Parameterizing the MARTE Metamodel

In this section, we describe our solution to enriching the MARTE (*Modeling and Analysis of Real-Time and Embedded systems*) metamodel with the support to parameterization. This metamodel has been defined in the context of the MARTE profile specification. The primary aim of this standard profile proposed by the Object Management Group (OMG) is to add capabilities to UML for the model-driven development of real-time and embedded systems. The concepts introduced in this profile significantly improve the usual way of modeling software and hardware platforms. Further extensions are provided to facilitate performance and scheduling analysis and to model platform services (e.g. services of an operating system). It is worth noting that, if MARTE defines a series of concepts to describe embedded systems, it relies on the UML specification to model the applications that will run on top of these systems. Finally, MARTE defines the concepts to model the deployment of the applications onto the systems.

5.1 Gaspard Modeling Process

Being a very generic specification for embedded and real-time systems, MARTE can be used in a multitude of manners. The use we make of MARTE is much more constrained, though, lying within the scope of the Gaspard Modeling Process [?]. The Gaspard Modeling Process is dedicated to intensive signal computation and allows the system-on-chip co-design. It takes in high-level UML models respecting the MARTE profile - in fact, these are the only input by users - and churns out code in different technologies: synchronous languages for formal validation, SystemC for simulation, OpenMP Fortran for execution of scientific computation and VHDL for circuitry synthesis. For this to be possible, the Gaspard2 framework provides different transformation chains, one for each target language. Presently, the transformations offer no support to parameterization. The idea is then to extend this support to beyond the UML-based input model and conserve generic components as long as needed within the transformation chains.

5.2 Overview of MARTE Metamodel

Although MARTE is primarily implemented in the form of extensions to UML, the UML profile for MARTE, the specification also includes a metamodel that defines MARTE concepts in a UML-independent fashion. In this report, we deal mainly with MARTE's General Component Model (GCM), the specific package that encompasses the component-based concepts.

Figure 6 represents the MARTE GeneralComponentModel package. A *StructuredComponent* specializes *BehavioredClassifier* to define a self-contained entity of a system, which may encapsulate structured data and behavior. Similarly to a UML *Classifier*, it owns properties, which can be attributes, or member ends of an association. A Property has a multiplicity - specified in terms of upper and lower bounds-, an aggregation kind and a type. The internal structure of a *StructuredComponent* can furthermore be referenced using the *parts* association, which points to *AssemblyParts*. *InteractionPorts* are a special kind of property owned by a structured component. An interaction port defines an explicit interaction point of the component with external elements. Two ports may be connected through an *AssemblyConnector*.

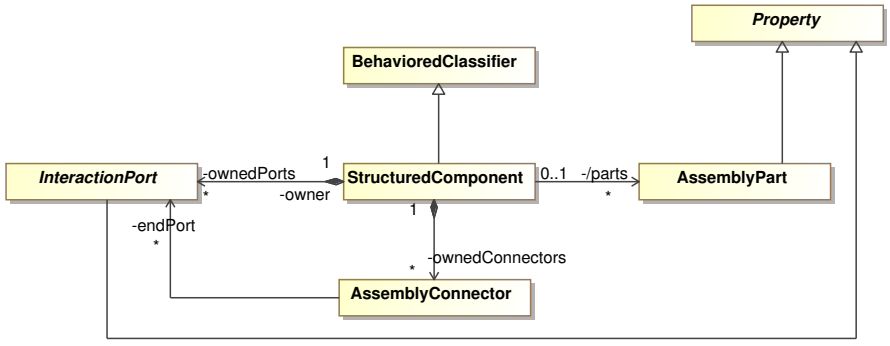

Figure 6: The Generic Component Modeling package of the MARTE metamodel

5.3 Extending the MARTE Metamodel

In this section we are going to explain how to connect our metamodel to MARTE. All we have to do is to indicate the elements that we want to "genericize" (i.e. elements that are allowed to receive parameters) and which elements we want to parameterize (i.e. elements allowed to have any feature referring to a parameter). This indication is achieved through inheritance relationships, as indicated in Figure 7. In this case, we would have:

- Elements that can be parameterized: *StructuredComponent* and its subclasses;
- Elements that can be exposed as parameters: *Properties* and their subclasses, InteractionPort and AssemblyPart;

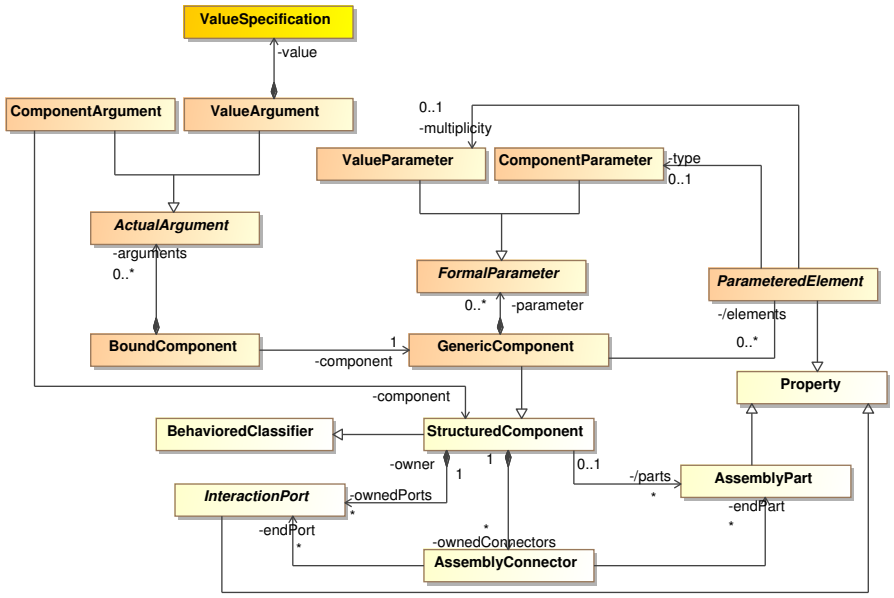

Figure 7: The Generic Component Modeling package of the MARTE metamodel after introduction of genericity

5.4 The Transformations

Once we have established the inheritance relationships shown in Figure 7, we also have to change our transformations accordingly. The idea is to make minimal changes to the existing transformations. Thus, the resulting transformation is, in fact, a two-step transformation chain (as shown in Figure 8). The first transformation takes in models in UML extended by the MARTE profile and generates objects from the Parameterized MARTE metamodel (obtained from the composition of the MARTE metamodel and the Generics metamodel, as shown in Figure 7), which we will call PM3. That is why a second transformation is needed, serving to substitute actual values for parameters and thus to remove

all references to the Generics metamodel. This second transformation will be detailed further on. All transformations[3] mentioned in this report have been formalized with the OMG Query, View and Transformations specification [?] and implemented with the QVT-O tool [?], from the EMF/M2M project. QVT-O is compliant with the Operational QVT specification.

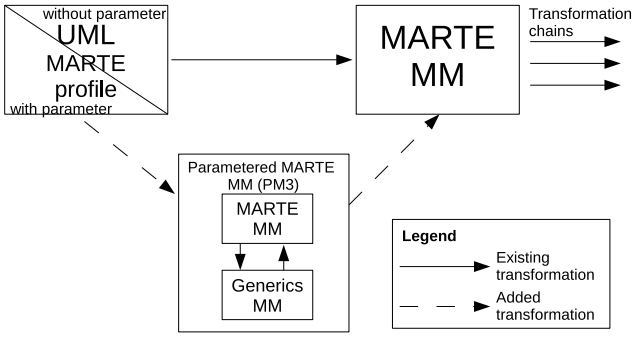

Figure 8: The new transformations

As a rule, elements from the UML template package will cause the transformation to create elements from the Generics metamodel. For example, a UML Component with a *UML::TemplateSignature* will give rise to a *Generics::GenericComponent* and a *UML::TemplateParameter* will give rise to a *Generics::FormalParameter*.

Now we would like to use the MARTE example to illustrate the impact of the parameterization process on existing transformations. Suppose there is already a transformation from UML to MARTE and we want to adapt it to take account of parameterization. After having set up the inheritance relationships between the Generics metamodel and MARTE, as illustrated in Figure 7, the existing transformation rules (i.e. QVT mappings) can be changed without too much effort. For example, let's say we have the following mapping to convert regular UML *Ports* into MARTE *Flowports*:

```
mapping UML::Port::oldToFlowport():GCM::FlowPort

{
   -- whatever mappings needed
}
```

All we have to do is to create a wrapping mapping that encompasses both the old mapping and the mapping to convert parameterized UML ports. The QVT 'disjuncts' feature will help us out with this task, as we see in the code below (note that the only modification to the old mapping is the introduction of the 'when' clause, responsible for filtering out parameterized UML ports):

```
abstract mapping UML::Port::newToFlowport():GCM::FlowPort
   disjuncts UML::Port::oldToFlowport, UML::Port::genericsFlowport
   {}

mapping UML::Port::oldToFlowport():GCM::FlowPort    -- the existing mapping
       -- no template parameter, so regular Port
   when{ self.type.oclAsType(Class).templateParameter.oclIsUndefined() }
{
   -- whatever mappings needed
```

[3]The transformations can be obtained from `http://www2.lifl.fr/west/gaspard/`

```
}

mapping UML::Port::genericsFlowport():Template::ParameteredElement
      -- specifies a template parameter, so a parameterized Port
  when{ not self.type.oclAsType(Class).templateParameter.oclIsUndefined() }
{
  -- parameter related mappings
}
```

Thus, the resulting mapping will take care of separating regular ports from parameterized ports and generating the adequate output elements. And this structure can be replicated to the other elements, though sometimes some adaptation might be required. With UML parts, for example (a part is a UML Property that is not a Port), things get slightly more complicated, since now there are three possible paths to follow in the transformation. If it is a parameterized part (i.e. one of its features refers to a TemplateParameter), then a *Generics::ParameteredElement* is generated and if it is a regular part, a simple *MARTE::AssemblyPart* will be created - exactly like the FlowPort example. However, a third alternative is required that addresses the case when the part name encodes an anonymous binding (i.e. something like `a<k→1,T→T1>`). In this case, two new elements will be generated: a *Generics::BoundComponent* and a *MARTE::AssemblyPart*. Figure 9 shows the resulting elements from applying this transformation to the model example taken from section 3.

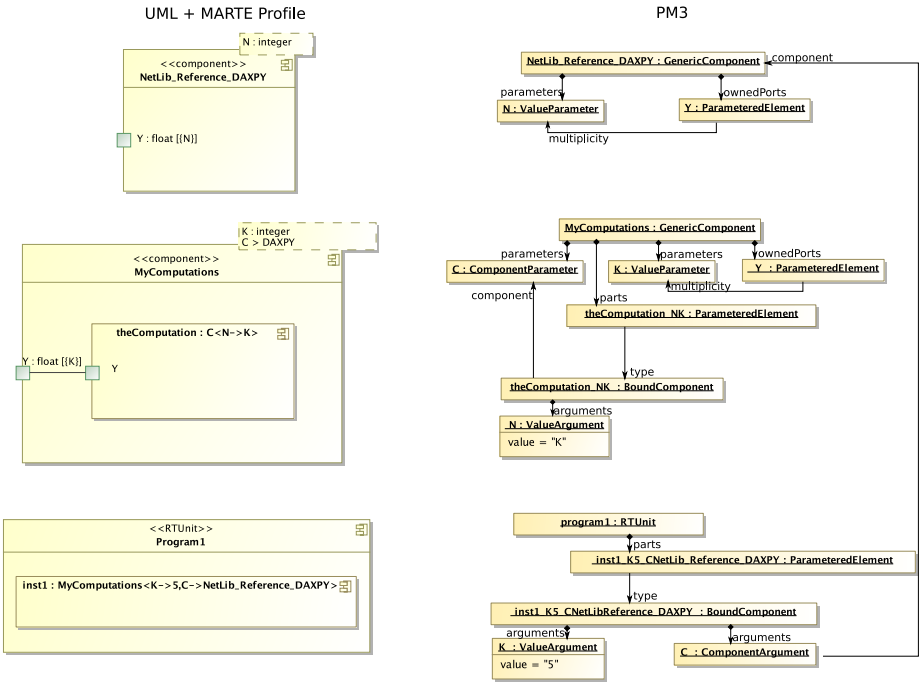

Figure 9: Transformation of the MARTE Profile to the PM3

The second transformation is responsible for generating the final MARTE model. Its main job is quite straight: it simply creates the final *MARTE::StructuredComponent* elements out of the *Generics::BoundComponent* elements - and the *Generics::GenericComponent* they

refer to (through the *component* reference). It is in this part that all parameters are replaced by the actual values, as previously mentioned in subsection 4.3. To wrap up, all old references to Generics elements are updated and dangling objects are removed. Figure 10 displays the final MARTE model, obtained after applying this transformation to the example taken from section 3.

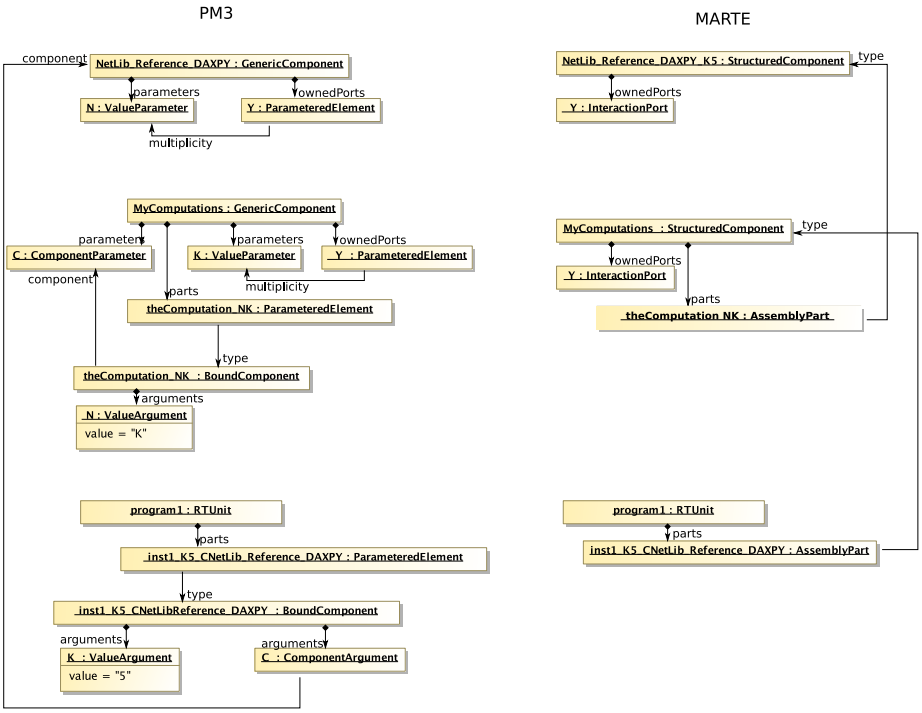

Figure 10: PM3 to MARTE

6 Conclusion

This report presented our studies concerning the parameterization of metamodels. We focused our attention on component-based metamodels. Three main advantages of parameterizing metamodels have been pointed at: constituting libraries of generic components, programmatically fine-tuning generic components and generating generic code. We proposed an approach based on a completely generic metamodel defining parameter-related concepts, which is taken to extend the metamodel to be 'genericized'. Such approach avoids touching target metamodels.

Moreover, we suggested a way of adapting existing transformations to take account of the newly added parameter-related concepts. As a case study, we applied our approach to the OMG MARTE metamodel for embedded and real-time systems. The transformations have been written in QVT, using the QVTO tool.

The next step in this work will be to define high-order transformations (HOT) that can be used to automatically adapt existing transformations to

take account of parameters. Since the adapted transformations are constituted of the fixed- (or boilerplate-) mappings dealing with the parameters-related concepts and of the old existing mappings, we think this follow-up work is highly viable.

Centre de recherche INRIA Bordeaux – Sud Ouest
Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex (France)

Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399