**Noname manuscript No.**
(will be inserted by the editor)

# Localized Model Transformations for Building Large-Scale Transformations

**Anne Etien**[1], **Alexis Muller**[2], **Thomas Legrand**[2] **and Richard F. Paige**[3]

[1] Inria Lille Nord Europe, LIFL CNRS UMR 8022,
Université Lille 1, France.
e-mail: `Anne.Etien@lifl.fr`

[2] GenMyModel, France
e-mail: `{Alexis.Muller, Thomas.Legrand}@genmymodel.com`

[3] Department of Computer Science,
University of York, UK.
e-mail: `richard.paige@york.ac.uk`

The date of receipt and acceptance will be inserted by the editor

**Abstract** Model-Driven Engineering (MDE) exploits well-defined, tool-supported modelling languages and operations applied to models created using these languages. *Model transformation* is a critical part of the use of MDE. It has been argued that transformations must be engineered systematically, particularly when the languages to which they are applied are large and complicated – e.g., UML 2.x and profiles such as MARTE – and when the transformation logic itself is complex.

We present an approach to designing large model transformations for large languages, based on the principle of *separation of concerns*. Specifically, we define a notion of *localized transformations* that are restricted to apply to a subset of a modelling language; a composition of localized transformations is then used to satisfy particular MDE objectives, such as the design of very large transformations. We illustrate the use of localized transformations in a concrete example applied to large transformations for system-on-chip co-design.

**Key words** Model Transformation, Reusable Transformation, Transformation Chaining

## 1 Introduction

Model-Driven Engineering (MDE) applies well-defined and tool-supported models and modelling languages to engineering problems [32]. A key component of MDE is the definition and application of model transformations, which transform source models into target models. These transformations are defined in terms of metamodels of source and target languages. Transformation enables many different approaches to system engineering [10], including code generation (e.g., through use of model-to-text transformations), stepwise refinement (e.g., through model-to-model transformations and refactorings), and viewpoint-based development, where different models (possibly expressed in different languages) are constructed and composed to provide a whole-system view.

Fundamental to these applications of MDE is the notion of *separation of concerns*. In MDE terms, this typically involves decomposing the MDE artefacts (i.e., models, metamodels and transformations) into parts. However, these artefacts are interdependent: models *conform* to metamodels; transformations are *defined* in terms of metamodels; and transformations *apply* to models. These interdependencies have been noted as a challenge for model-based approaches to system engineering [6,12]. In principle, separation of concerns in MDE is (like in other software engineering disciplines) helpful and necessary to manage complexity; in practice in MDE, it introduces new challenges that require specialized techniques to manage the dependencies between artefacts.

It has been argued [18] that model transformations, like other software engineering artefacts, must be systematically designed and implemented; given the importance of transformations in MDE, this is an understandable argument. As MDE is applied more widely, and large and complicated languages are used – such as UML 2.x and profiles such as SysML or MARTE – large transformations are more likely to be developed; examples have been published of transformations totalling tens of thousands of lines of code. Such transformations have substantial drawbacks [27], including reduced opportunities for reuse, reduced scalability, poor separation of concerns, limited learnability, and undesirable sensitivity to changes. Thus, it is desirable to *decompose* transformations, much as engineers decompose other artefacts (like architectures, components, object-oriented designs) into compositions of smaller transformations [27, 39]. Particularly, Vanhooff et al argue [39] that "*chaining many small transformations that each manipulate the model with one specific concern* [...] *allows [one] to better modularize the transformations themselves and as a consequence make individual transformations easier to implement and reuse.*" Other research has also argued that focusing on the engineering of transformations, and improving scalability, maintainability and reusability of transformations, is now essential, to improve the uptake of MDE [27,41] and to make transformations practical and capable of being systematically engineered [7].

This paper addresses the challenge of how to reduce the complexity of transformations while improving their reusability, modifiability and understandability. The solution we propose is independent of any transformation language and any business domain. The main contribution of the paper is to apply the principle of separation of concerns to model transformations, and introduce the concept of *localized transformations*, which are transformations that apply to a (typically very small) subset of an input metamodel of a transformation. Each localized transformation is designed and implemented to accomplish a specific transformation task, and involves a small number of concepts in the source language[1]. *Chains* of localized transformations are composed to satisfy whole-system transformation objectives. A novelty with our concept of localized transformation is *implicit copying*: those elements of the source model not in the context of a localized transformation are implicitly copied to the target of the transformation. This approach is related to the notion of *conservative copying* used in model migration [30]; a detailed comparison between our approach and conservative copying appears in the related work section of this paper.

The concept of localized transformation is valuable when designing a *family* of related transformations that exhibit similarities and variabilities, e.g., when mapping from a core source language (such as UML or MARTE) to different technologies. In such situations, the family consists of a set of chains of transformations, where each chain is different but a number of steps – each consisting of a localized transformation – are shared. Our concept of localized transformation aims to enable their reuse in different transformation chains.

This paper is organized as follows. Section 2 presents the advantages of decomposing transformations and introduces more precisely the concept of localized transformation. Section 3 details how localized transformations can be implemented. Section 4 illustrates the reusability of localized transformations with examples from the transformation chains of the Gaspard environment [17] for building embedded systems using MDE. Section 5 describes related work on transformation composition and reusability. Section 6 draws some conclusions and additional perspectives, particularly on the limitations of localized transformations, and suggests ways in which some of these limitations can be addressed.

[1] This is analogous to *components* in system design: a component has a precise interface and is used to encapsulate functionality for particular tasks.

## 2 Towards managing transformation complexity

MDE and its various flavours involve the design and implementation of model transformations. As larger systems are designed using MDE, larger and more complex transformations will result; as such, systematic approaches to managing transformation complexity are needed. *Decomposition* is a proven technique in managing complexity in many domains (including complexity in MDE, e.g., for defining more modular languages [42]). Decomposition is also applicable in the context of designing transformations: instead of designing and implementing a single large transformation, the principle is to compose a sequence of smaller, ideally simpler transformations that accomplish at least the same tasks. Figure 1 illustrates a traditional way to decompose a transformation **T**, from the metamodel $MM_i$ to the metamodel $MM_o$, into a set of smaller ones, $T_1$, $T_2$ and $T_3$, through the introduction of a set of intermediate metamodels $MM_1$ and $MM_2$.

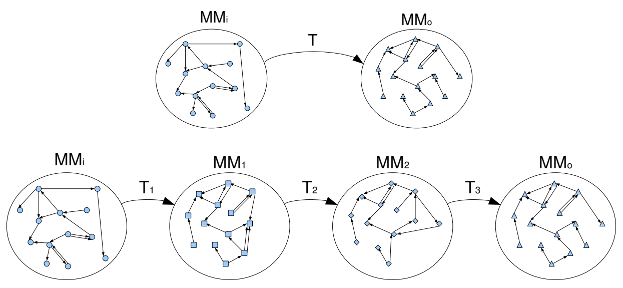


**Fig. 1** Decomposition of a complex transformation T into smaller ones

A side-effect of such a decomposition (or *chaining* of transformations) is that there are increased numbers of dependencies between increased numbers of artefacts. New, intermediate metamodels must be introduced (e.g., $MM_1$ in Figure 1), new models are produced, and the overall amount of coupling in the entire transformation increased. In essence, the complexity, implicit in the original large transformation **T** in Figure 1, is now made *explicit*, expressed in the intermediate metamodels and the intermediate small transformations in the chain. In order to better manage this complexity and increase the maintainability, the input metamodel (representing the whole system of interest) can be decomposed into smaller parts. For each part, a transformation would be designed; each such transformation would have a specific purpose. In the process, intermediate metamodels are likely to need to be introduced. Each intermediate metamodel would be a subset of the input metamodel (with additional concepts created, or existing concepts removed by other transformations). Transformations would be extended in order to cover the entire input metamodel, and then chained to satisfy the overall requirements of the transformation problem. The order in which transformations are chained is determined according to those concepts that need to be introduced or removed in the metamodels. Figure 2 summarizes the main principles of localized transformations, as discussed precisely in Section 2.3 (the figure does not attempt to capture the sequentiality of execution, though does attempt to represent the restricted context and limited size of each localized transformation).

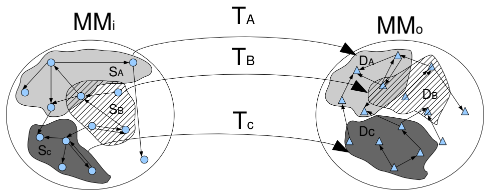


**Fig. 2** Localized transformations

Informally, a localized transformation is analogous to a notion of *component* in system design. A component encapsulates a restricted set of functionality and defines a precise interface. A localized transformation encapsulates a restricted set of functionality (for mapping source to target concepts) and defines a precise interface (the subsets of the source and target metamodels of concern).

An issue that we have not discussed is *concrete practices* for decomposing a large transformation, and how to choose between different decompositions. We do not attempt to present concrete decomposition practices in this paper; such a question is similar to "how do you decompose a system into subsystems?" Answering such questions requires a comprehensive body of knowledge – e.g., a set of proved decomposition patterns – and given the relative immaturity of the field of model transformation engineering, such a body of knowledge is not yet available. As such, choosing good decompositions, and deciding how to chain transformations together, is up to the judgement of the transformation engineer. However, there are several points that can be made to help decide on how to choose a decomposition:

- As with system decomposition, transformation decomposition should be carried out keeping in mind the ultimate end-use of the transformation: is it for humans to read (and review)? Is it done so that the transformation execution engine can more easily optimize execution? Is it to maximize reuse?
- The granularity of each individual transformation is a critical point of concern. If individual transformations are too large (e.g., measured in terms of lines of logical code or elements in the input and output metamodels), they may be too difficult to understand, manage, reuse and evolve. If they are too small, their reuse may be of limited value.
- The logical complexity of each individual transformation is also important, for understandability and for purposes of simplifying verification and validation. If transformations attempt to carry out too much computation (e.g., measured in terms of satisfying requirements/goals), they may be too difficult to understand, manage, reuse, or validate. If they carry out too little computation, they will be of limited value in reuse.

Simply identifying useful decompositions of a large transformation is insufficient; we also need useful ways to manage the interdependencies that arise as a result of the decomposition. It is for this reason that we have developed the notion of localized transformation. To help motivate and clarify this notion, we first introduce quality attributes dedicated to transformation chains and provide a motivating example.

## 2.1 Quality of Transformation Chains

In [37], the authors describe quality attributes and associated metrics to evaluate the quality of model transformations. In order to better motivate our approach, we adapt some of these quality attributes to transformation chains. [37] proposes a large number of metrics; we describe only those that we claim to affect via our approach.

*Understandability.* The amount of effort required to understand a transformation chain. Understandability is related to modifiability and reusability. The easier it is to understand a transformation chain, the easier it should be to modify or reuse.

*Modifiability.* The extent to which a transformation chain can be adapted to provide different or additional functionality. The main reason for modifying a transformation chain is changing requirements. Another reason is that the targeted language may be subject to changes. Modifiability captures the amount of effort needed to modify a transformation chain such as to deal with changes in either the requirements, or the source or target metamodel.

*Reusability.* The extent to which (a part of) a transformation chain can be reused as-is by other transformation chains. Reusability is different from modifiability, which refers to modifying a model transformation.

*Reuse.* Reuse is the counterpart of reusability. Reuse is the extent to which a transformation chain reuses parts of other transformation chains.

*Modularity.* The extent to which a transformation chain is systematically structured. With systematically structured we mean that every transformation in a transformation chain should have its own purpose. Modularity is related to reusability. If functionality is distributed over transformations it is more likely that parts of it can be reused for other transformation chains. Therefore, the size of transformations is also an important aspect of modularity.

The next subsection introduces a motivating example illustrating how our approach could help to enhance quality of transformation chains.

### *2.2 Motivating Example*

Our motivating example focuses on the generation of a traditional information system from a UML model, where the information system requires use of the classic model-view-controller (MVC) architecture. The generation process requires steps to generate the view and controller classes associated to each model class; to create the methods to create/update/delete the business objects and access their attributes; to describe the links between the objects and the database; to create the database (if it does not already exist) directly or using tools like hibernate; to generate code, etc.

It would be possible to generate the code in one step from a UML class diagram that captures the system functionalities and business logic. However, reusability, modularity and modifiability of the chain would be very low. Indeed, if a new technology or a new language is thereafter to be targeted, almost nothing of the generation process would be reusable. Introducing one or several intermediate metamodels, and thus intermediate transformations, would help to reduce the complexity of the one-step generation process, and would increase the modularity and the reusability (as in Figure 1).

Usually, chain developers define the intermediate metamodels that they need, and then define the transformations. Thus, in our motivating example, they would define, for example, an MVC metamodel that would move the existing classes into a business package, would introduce Control and View concepts, would link them together, and would connect them with classes of the original metamodel corresponding to the business concepts. This metamodel would be very similar to that of UML; it may even be an extension thereof. Such an MVC metamodel is approximately two hundred and fifty concepts (and as such we do not present it). The transformation from UML to the MVC metamodels would be primarily dedicated to *copying* of an element from the input model to the output model; such corresponding concepts are considered to be semantically equivalent. Given the size of the metamodels involved, such a transformation could be significantly complex if it covers the whole input metamodel, and the *intent* of the transformation (*i.e.* introduce Control and View concepts) would be lost admidst the functionality that carries out the copying. The understandability and the modularity of the chain would be limited. Alternative designs would be to refactor the UML model (and thus not introduce Control or View instances but to consider instances of Class as such) or to enhance the intermediate (MVC) metamodel with other concepts (e.g., using a Data Transfer Object) that should be added before generating the code. This could be done in order to introduce more than just Control and View to the business objects, and could, for example, link the business objects and the database. In the first case, the generation code would be difficult to produce since the Control and View concepts would not be explicit. In the second case, the difference between the UML and the intermediate metamodels would be more substantial and thus the requisite copying functionality would be reduced. However the overall complexity would increase and reuse of the transformation would become more difficult.

Stepping back, consider a component-based approach to software design: building a system using a component-based approach does not require that a first version of the system already exists. Given requirements, a system specification can be produced through decomposition and implementation of new components, as well as composition of pre-existing components of different sizes, complexities and granularities. We are effectively proposing an analogous approach for *transformation design*: when building a complex transformation, we should be able to choose from pre-existing transformations (of different sizes, complexities and granularities) in constructing a transformation chain. But, in order to support this component-based process, we have identified a new type of transformation – localized transformations – that help to manage the complexity inherent in such a process.

Returning to our conceptual example, the chain developer would consider the different steps required to target the technology and generate the corresponding code (*e.g.* create the methods to create/update/delete the business object or introduce the getter and setter methods). Each of these steps would correspond to a localized transformation. These localized transformations are independent of any business domain and targeted technology. They can be reused in different transformation chains. The metamodels are defined for each transformation by identifying the involved/created/modified/-deleted concepts.

The following subsections respectively introduce the concept of localized transformation and the mechanisms by which they can be composed.

### 2.3 Localized Transformations

A localized transformation is a specification of a model-to-model transformation that possesses the following properties:

1. **Metamodels overlapping:** its input and output metamodels must *overlap*; some metaclasses may belong both to the input metamodel and the output metamodel. Consequently, the elements of the input models are not all consumed and the elements of the output models are not all produced by the localized transformation. Some elements may be only read or modified, and thus exist both in an input model and the corresponding output model.
2. **Restricted context:** it applies to a precisely defined and restricted part of the input model; more precisely, we say that a localized transformation has a precisely defined *context* that is a subset of an input metamodel. A localized transformation aims to be extended with the `Extend` operator [14] and be applied on models conforming to the whole input metamodel, even if it is defined on one of its subparts.
3. **Conceptual integrity:** its behaviour establishes a unique and specific intention.

In other words, a localized transformation applies to a tightly prescribed, typically small-in-context part of an input model; all other parts of the input model are not affected or changed by the localized transformation. Conversely, the metamodels involved in a localized transformation contain only those elements *required by the transformation itself*. An extension mechanism allows engineers to extend the input and output metamodels of a localized transformation, so that the input model conforms to the actual metamodel. When executed, the localized transformation performs a single task, e.g., it satisfies a unique requirement, it provides a unique unit of functionality. In our motivating example, the contexts of the various transformations have been clearly identified (*e.g.* introducing MVC concepts, or adding a getter and a setter for each attribute).

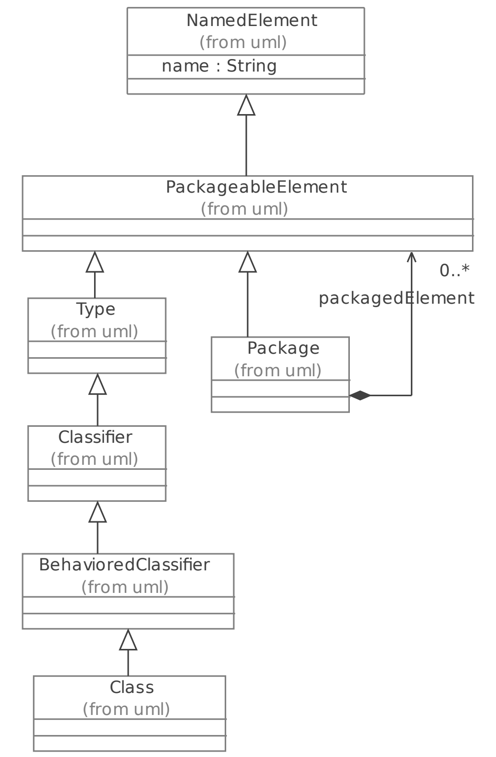
The input metamodel may be different from the output metamodel (and as such, localized transformations need not be update-in-place transformations). Some concepts in the input model will be repeated in the output model, i.e., they will simply be copied over from input to output model, analogous to [30]. Manually writing such transformation logic is tedious and error prone; moreover, in the case of complicated transformations, such logic (which may be repeated in different parts of a chain of transformations) increases interdependencies and can reduce reusability. Thus, to increase flexibility we distinguish two parts of a localized transformation: the part that captures the essential transformation logic, and the part that copies that subset of the input model to the output model. In this manner, a localized transformation can be specified with small (intermediate) metamodels only containing the concepts used and affected by the transformation. The extension mechanism, combined with the implicit copy, provide the means to manage the transformation engineering process.

For the example in Section 2.2, the transformation that generates the MVC architecture is defined on small metamodels. Figure 3(a) represents the input metamodel. It gathers only the UML concepts useful to the transformation. The output metamodel introduces the concepts of *View* and *Control* and their relationship with the *Classes* that in the future model correspond to the business classes. It is represented in Figure 3(b). In this example, it is coincidence that the concepts introduced in the target metamodel all reference existing concepts via unidirectional relationships (e.g., generalization or directed association); however, localized transformations impose no such restrictions. The modification of existing concepts (like in any other operation in a transformation) can only occur if the information can be deduced from existing ones.

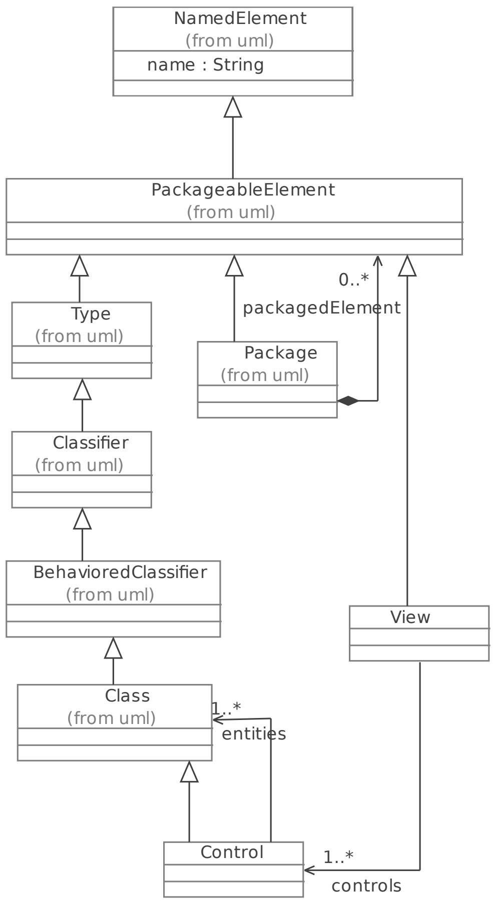
A model transformation, therefore, is conceptually constructed of a set of localized transformations (whose input metamodels are small subparts of the input metamodel of the transformation), as well as a set of functions that manage the *copying* of concepts outside of the context of each localized transformation. The copying functions do not need to be specified or generated[2], copying is handled automatically through use of the `Extend` operator.

In [14], we formally defined the `Extend` operator, which was introduced to extend input and output metamodels of transformations in order to make them compatible and chainable. Used with localized transformations, this operator preserves their behavior and uses the identity to process elements not involved in the original transformation. Concretely, the `Extend` operator enables writing each localized transformation as an *inout* transformation involving a small number of concepts. Associated with a copy of the input model as a whole (in practice, the file corresponding to the model is copied, whereas in [30] each element is separately copied), those concepts impacted by the localized transformation are modified, added or removed, the other remaining exactly the same, thanks to the implicit copy. In other words, this operator extends the notion of *inout* transformation to transformations where input and output metamodels may be different, but the copy/identity function is implicit. Concretely, the input file is copied and the transformation is executed as an *inout* transformation on this copy (see Section 3 for more implementation details). More formally, let $t$ be a localized transformation from the metamodel $S_A$ to the metamodel $D_A$ ($t : S_A \to D_A$) and $MM_i$ an ordinary metamodel. $\texttt{Extend}_{MM_i}(t)$ is a

[2] Unlike, e.g., in higher-order transformation-based approaches like `http://www.eclipse.org/atl/atlTransformations/#KM32ATLCopier`.



(a) Input metamodel.



(b) Output metamodel.

**Fig. 3** Input and output metamodels of the MVC transformation

transformation $T$ from the $S_A \cup MM_i$ metamodel[3] to the metamodel $MM_o$ ($T : MM_i \cup S_A \to MM_o$), having the same behaviour than $t$ such as:

- $MM_o = D_A \cup (MM_i \setminus S_A)$, where $MM_i \setminus S_A$ is the part of the metamodel $MM_i$ that was not involved in the transformation, *i.e.* the part whose instances are implicitly copied
- $T(m) = t(m)$ if $m$ is a model instance of the metamodel $S_A$, applying $t$ or its extended version $T$ is exactly the same when $m$ conforms to $S_A$
- $T(m) = m$ if $m$ is a model instance of the metamodel $MM_i \setminus S_A$, $m$ is simply copied since it does not contain elements instantiating a concept of $S_A$
- $T(m) = T(n) \cup (m \setminus n)$ with $n$ the part of the $m$ model typed by $S_A$. $T$ is composed of two parts, the transformation $t$ and the copy.

$(D_A \setminus S_A \neq \emptyset)$ implies that $t$ and thus also $T$ introduce new concepts potentially not in $MM_i$; correspondingly, $(S_A \setminus D_A \neq \emptyset)$ means that some concepts have been consumed by $t$ and thus also $T$. A concept is consumed by a transformation if it exists in the input metamodel of the transformation but not in the output metamodel. Thus, the execution of the transformation aims to remove all instances of those concepts present in the input model. In theory, it is always possible to choose $MM_i$ such as $S_A$ is included in $MM_i$; however, in practice, $MM_i$ is not arbitrarily chosen, it depends on the chain and corresponds to the input metamodel of the chain plus (*resp.* minus) those concepts introduced (*resp.* removed) by other transformations. Indeed, if $S_A$ is not a subpart of $MM_i$, this means that some concepts are useful to the execution of the transformation $t$ but no instance will be found in any input model: another localized transformation introducing these concepts must be executed beforehand. Finally, extending $t$ with various metamodels $MM_i$ enables to easily reuse $t$.

The notion of localized transformation is an addition to the classifications established in [9,21], where the input and output metamodels are different, but with some concepts in common. The transformation copies the input model and then modifies it to produce the output model. Indeed, Mens *et al.* distinguish heterogeneous transformations, where the input and the output metamodels are different, from endogenous transformation defined on a unique metamodel. Czarnecki differentiates approaches mandating the production of a new model from nothing, from others modifying the input model (*e.g.* in-place transformation). These classifications do not consider sharing and copying with potentially different input and output metamodels inherent in

[3] We adopt the metamodel, model and conformance definitions established by Alanen *et al.* [1]. From these definitions, the union, the intersection and the difference are defined on both metamodels and models respectively as the union, the intersection or the difference of each set defining the metamodels or the models.

localized transformations. The notion of localized transformation is therefore a new contribution to this taxonomy. In terms of implementation, approaches that come closest to localized transformations are those where a source and target metamodel are merged (i.e., using the union operator of [1]), an exogenous transformation applied, then a query is used to filter out those elements of the source metamodel that do not appear in the target metamodel. While this approach achieves the same result as localized transformations, our approach does not require construction of queries, and abstracts from the above-mentioned sequence of steps (which are encapsulated in the `Extend` operator and our transformation engine). Moreover, our approach also aims to treat transformations like software components in system engineering, as discussed earlier in this section.

Returning to our motivating example, the MVC transformation uses the metamodels presented in Figure 3 respectively corresponding to $S_A$ and $D_A$. It aims to associate a Control and View instance to each instance of Class. The MVC transformation is applied on a model instance of the UML metamodel (playing the role of $MM_i$) and produces a model instance of the ExtendedMVC ($MM_o$, *i.e.* the merge of the UML metamodel and the MVC metamodel of Figure 3(b)). The UML concepts that are not impacted by the transformation are copied.

Such an approach can help to reduce the complexity of the transformations and can also enhance their maintainability, since each transformation has a specific purpose, is a result of limited logical complexity, and is applicable to a limited number of modelling concepts. The validity of a localized transformation may, as a result, be easier to check since its purpose is clearly identified (*e.g.* adding Control and View to classes for the MVC transformation) and its size and complexity reduced. Localized transformations can also enhance reusability, because their input and output metamodels only contain the concepts affected by the transformation. Since such transformations are also associated with a function managing the copy, they can be performed on any metamodel containing their input metamodel. Such cases may frequently happen in the context of MDE-based design environments targeting various implementation languages, or in those targeting the development of software product lines. For example, the MVC transformation can be used in chains targetting J2EE or .Net technologies, service oriented architecture or more traditional architectures. We foresee attempting to relax the strict inclusion constraint in the future, in order to use localized transformations with generic model transformations [8, 29].

Additionally, an approach based on localized transformations has the potential to allow more independent design of transformations. By separating concerns in the transformation problem into smaller sub-problems, each sub-problem (and hence, each localized transformation to be constructed) can be designed and implemented independently.

### *2.4 Composition of Localized Transformations*

Once individual localized transformations have been defined, they must be composed in order to form a transformation chain. Defining this composition is non-trivial: while the input metamodel for the chain is known, the order in which localized transformations are executed has to be calculated precisely, since some orderings will not lead to a result that conforms to the output metamodel (e.g., by leaving an intermediate model in an inconsistent state that cannot be reconciled by any successive subchain of localized transformations).

In [14], we formally defined rules to identify valid compositions of transformations. These rules can be applied to localized transformations, since they consider transformations whose input and output metamodels overlap. They rely on a structural analysis based on types of the small metamodels involved in each localized transformation. We briefly summarise this here.

Consider two localized transformations, $t_A : S_A \rightarrow D_A$, a transformation whose input metamodel is $S_A$ and the output $D_A$, and $t_B : S_B \rightarrow D_B$ whose input and output metamodels are respectively $S_B$ and $D_B$.

*Definition: Chaining of localized transformations.* $t_A$ and $t_B$ two localized transformations, can be chained if there exists a metamodel $MM_A$ on which the first transformation can be extended using the `Extend` operator and if the concepts used by the second transformation are included in the output metamodel of the first extended transformation. More formally, $\exists\ MM_A$ such as $S_B \subseteq D_A \cup (MM_A \setminus S_A)$. This inclusion implies that the concepts used by the second transformation ($t_B$) are not consumed by the first one ($t_A$) *i.e.* $S_B \cap (S_A \setminus D_A) = \varnothing$.

From this definition, it is possible to deduce the following property:

*Property: Chaining of extended transformations.* If $t_A$ and $t_B$ can be chained then their extension version $T_A = Extend_{MM_A}(t_A)$ and $T_B = Extend_{D_A \cup (MM_A \setminus S_A)}(t_B)$ can also be chained corresponding to the classical $T_A \circ T_B$.

As illustrated in Figure 2, the input metamodels $S_A$ and $S_B$ (which we term *contexts*) are subsets of $MM_i$ with $MM_i = MM_A \cup S_A$ (plus eventually other concepts created by other localized transformations). Three cases may occur:

1. *$t_A$ and $t_B$ can only be combined in one order ($t_A$, $t_B$ for example).* This means that $t_A$ can be chained with $t_B$ or $t_B$ can be chained with $t_A$.

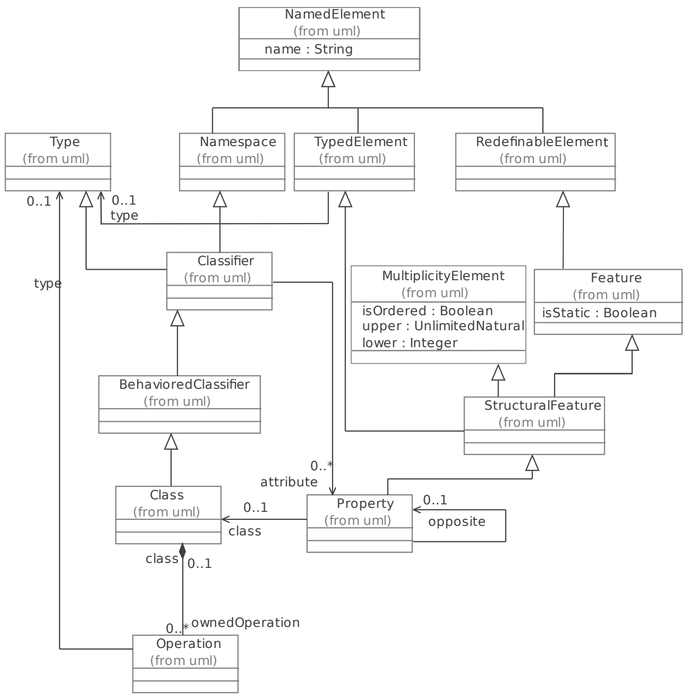
2. *$t_A$ and $t_B$ can be combined in both orders.* The order of the two localized transformations $t_A$ and $t_B$ can be swapped if both $t_A$ can be combined with $t_B$ and vice-versa *i.e.* the chaining definition is applied in both orders. But this cannot be guaranteed for the models. If the input metamodels of the two transformations $t_A$ and $t_B$ have no common elements and if the concepts required by $t_A$ (respectively $t_B$) are not produced by $t_B$ (respectively $t_A$), they can be combined and the resulting model does not depend on their execution order. If $(S_A \cap S_B) = \varnothing$ and $(D_A \cap S_B) = \varnothing$ and $(D_B \cap S_A) = \varnothing$ then, for all models $m$, chaining extended versions of the transformations $t_A$ and $t_B$ leads to the same result than chaining them in the opposite order.
3. *$t_A$ and $t_B$ cannot be combined at all.* The combination of $t_A$ and $t_B$ transformations is impossible when each transformation consumes concepts useful for the execution of the other *i.e.* if $S_B \cap (S_A \setminus D_A) \neq \varnothing$ *and* $S_A \cap (S_B \setminus D_B) \neq \varnothing$.

The reader is referred to [14] for more specific details. Any analysis based on instances such as concepts just read and not consumed or negative application conditions are not yet taken into account. There are included in our future work.

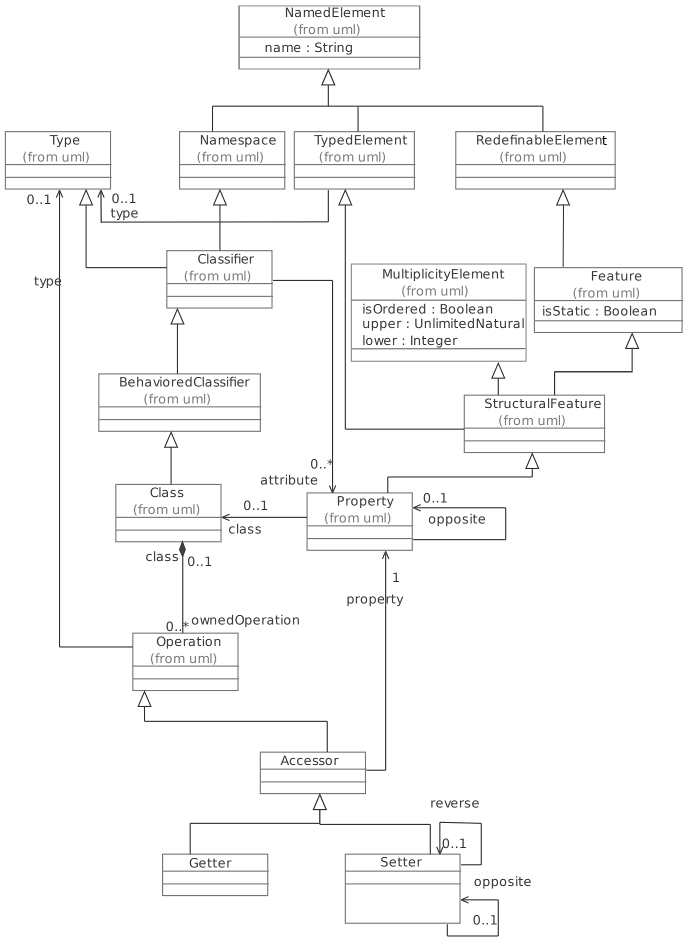
In our motivating example, $MM_A$ is the UML metamodel, $S_A$ and $D_A$ are the metamodels associated to $t_A$, the MVC localized transformation and presented respectively on the top and the bottom of Figure 3. $t_A$ only introduces concepts, *Control* and *View*. $t_B$ is the localized transformation introducing the getters and the setters for each property. The metamodels $S_B$ and $D_B$ associated to this transformation are shown in Figure 4 (respectively on top and on the bottom of the figure). $S_A$ and $S_B$ are strictly included in the UML metamodel ($MM_A$). Moreover, both of these transformations only add concepts. Each concept of $S_B$ exists either in $D_A$ or in UML 2.0 and are not removed by $t_A$. The input metamodels are included in their respective output ones; $(S_A \setminus D_A) = \varnothing$ and $(S_B \setminus D_B) = \varnothing$. The following equalities are true:

– $S_B \subseteq D_A \cup (MM_A \setminus S_A)$ and
– $S_A \subseteq D_B \cup (MM_A \setminus S_B)$;

$t_A$ and $t_B$ can be chained in both orders. However, $(S_A \cap S_B) \neq \varnothing$ because some concepts *e.g.* *Type* or *Class* are present in $S_A$ and in $S_B$. Thus, it is not possible to ensure that the chaining $t_A$ and $t_B$ or $t_B$ and $t_A$ leads to the same results whatever the input models. For the moment, the chain designer has to check which order is the one required. We plan, as future works, to provide more automated support for representing and selecting an ordering. In this case, both orders are correct since the MVC transformation does not introduce any property. Thus, whatever the order, the getters and setters will be introduced for each property by executing $t_B$.



(a) Input metamodel.



(b) Output metamodel.

**Fig. 4** Input and output metamodels of localized introducing the getters and setters

## 3 Implementation

This section presents LTDesigner, the transformation chain orchestration tool included in the Gaspard framework dedicated to embedded system design and briefly described in section 4. LTDesigner is independent of any application context. It is a platform based on Eclipse that uses the Eclipse Modelling Framework (EMF) for modelling and metamodelling. It implements the `Extend` operator and executes localized transformation chains. The main idea for the implementation is to support the localized transformations through in-place transformations, though as we have discussed previously, localized transformations are applicable to any transformation framework or tool that supports *inout* transformations.

### *3.1 Implementation of the `Extend` operator*

As previously explained, localized transformation adopt the same principle as *inout* transformations to manage the implicit copying of model elements that are not in the context of the localized transformation. However, unlike *inout* transformations, localized transformations may be written where input and output metamodels are not identical. The key principle behind our implementation is to artificially (and automatically) enhance the metamodels involved in a localized transformation so that they are identical and then perform an *inout* transformation. Figure 5 sketches this strategy. In this figure, the localized transformation from $S_A$ to $D_A$ is considered. It is extended, using the `Extend` operator, to be applied to model instance of $MM_i$. $S_A$ is included into $MM_i$. More precisely, the $MM_i \cup D_A$ and $MM_i \cup S_A \cup D_A$ metamodels result from merging the different metamodels based on the names of the metaclasses.

In order to simulate the model-to-model mode (the transformation is written using the *inout* mode), the file corresponding to the input model is first copied, in order to, upon termination of the execution of the transformation, separately preserve the input and the output models. The input model (instance of the metamodel $MM_i$) is considered to be an instance of the virtual metamodel $MM_i \cup D_A$ (step 1 in Figure 5). Concretely, the reference of the input model to the $MM_i$ metamodel is replaced by a reference to the virtual metamodel $MM_i \cup D_A$. The model can then be transformed by the localized transformation using the classical *inout* mechanisms with $MM_i \cup S_A \cup D_A$ as the unique input/output metamodel (step 2) (*i.e.* the input model conform to the $MM_i \cup S_A \cup D_A$ is modified in-place). Finally, another metamodel evolution, from $MM_i \cup S_A \cup D_A$ to $D_A \cup (MM_i \setminus S_A)$, is performed on the resulting model to remove concepts consumed by the transformation (step 3). This last step is the most delicate. To be error-free, the transformation must conform to its definition, *i.e.*, it must actually have removed all model elements that are instances of removed concepts.

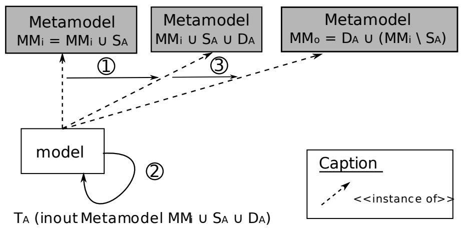


**Fig. 5** Schema of the `Extend` implementation

### *3.2 Model transformation chain engine*

Our engine also supports the specification and execution of transformation chains. For this purpose, LTDesigner uses the component paradigm. Each transformation is represented as a component, as is each of its parameters (either in, out or inout) that corresponds to a metamodel. Chains are built by linking output parameters to input ones through connectors (see section 4.2.3 and more precisely Figure 10 for an example). If linked parameters do not have the same type, the LTDesigner transformation engine automatically performs a metamodel extension. Thus, the order of the transformations has to be carefully determined based on the composition rules described earlier, in section 2.3. Each localized transformation may introduce or consume concepts. Thus, by successively extending the transformations, potential chaining errors are reduced since typing constraints can be checked. This would not be the case if the transformations would have been executed on a global metamodel containing the source metamodels of all the transformations.

LTDesigner enables the developer to create chains from off the shelf localized transformations by providing a transformation chain run-time and a mechanism to build chains. If LTDesigner has been designed independently from any application domain, the chain run-time is yet available only as a subpart of Gaspard. The transformation chain interface has been used by the Gaspard developers but is not provided to the Gaspard users since the chains are built once for all and can not be modified by the Gaspard users.

Figure 6 presents a layered view of the Gaspard architecture. The user level provides a dedicated interface to design UML models enhanced with the MARTE profile, as well as a sample model library and a documentation explaining the way to build such models. The superstructure gathers the metamodels, the localized transformations and the chains. These two levels are dedicated to one business domain, embedded systems in the case of Gaspard, but could be replaced by corresponding levels adapted to another domain. The infrastructure level corresponds to the LTDesigner modules and a traceability engine. The basis is provided by Eclipse and plugins. Currently, the LTDesigner engine supports transformations implemented in Java and QVT, *i.e.*, is able to launch Java or QVT transformations and also can save the intermediary models. The implementation of the `Extend` operator is not included in the launchers and is thus independent of any transformation language. Nevertheless, the `Extend` operator and thus the localized transformations require a language having an *inout* mode supporting the modification and removal of existing elements. Such a property is common in existing transformation languages (for example, ATL, Epsilon, QVT and Henshin all support it in different ways). We plan to extract LTDesigner from Gaspard and to provide it as a standalone tool to compose chains from transformations available in libraries whatever the application domain (*e.g.* information system or embedded system).

The next section shows examples of such transformation chains.

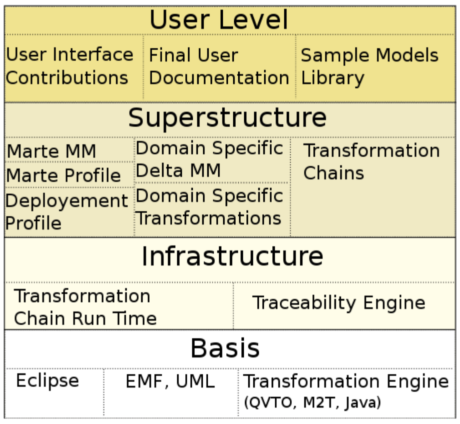


**Fig. 6** Schema of the Gaspard architecture

## 4 Examples

This section demonstrates aspects of the localized transformation approach in action, via examples taken from a realistic case study. We describe several transformation chains that can be constructed from localized transformations that have been collected in a transformation library within the Gaspard environment. Overall the localized transformations are designed to support the construction of transformations from the MARTE profile of UML [26] to platform-specific implementation languages, namely SystemC, OpenCL, pThread and OpenMP. The transformations from MARTE are built through reuse of a number of chained localized transformations.

Subsection 4.1 briefly introduces Gaspard business concepts. Subsection 4.2 provides examples of transformations and the chains that implement them.

### *4.1 Context: the Co-Design Environment Gaspard*

Gaspard[4] is a hardware/software co-design environment dedicated to high performance embedded systems based on massively regular parallelism [17]. In this environment, the designer separately specifies the application, the hardware architecture and the mapping of the former to the latter at a high level of abstraction. The designer uses the MARTE profile of UML (which supports real-time embedded system concepts). From these high level specifications, code for high performance computing, hardware-software co-simulation, or hardware synthesis is automatically produced by model transformation chains constructed from localized transformations.

*4.1.1 High Level Concepts Used in Gaspard* In Gaspard, when modelling an application, a component corresponds to a task, whereas when modelling an architecture, a component corresponds to a hardware resource, *e.g.* a processor or memory. A hundred conceptual tasks relating to exactly the same action, but realized with different data, are modelled by a unique task. The way that the data are consumed by each task is specified by the *LinkTopology* concept (*Tiler* or *Reshape*) associated to an *AssemblyConnector*. Similarly, 64 identical processing units are represented by only one with a repetition-value set to 64. Parallelism in the application and in the architecture is thus concisely expressed.

Gaspard adopts a component-based approach. Thus, a task may be complex and composed of several others; it may also be organized in a hierarchy. In MARTE, a composed task is represented as a *StructuredComponent*, whereas the contained ones are instances (*AssemblyPart*) referring to another externally defined component.

A task can be mapped onto processing units through the *TaskAllocation* link. Data are similarly mapped onto memory with the *DataAllocation* linking the port to the memory.

Figure 7 sketches the concepts of MARTE used in Gaspard. Further details on Gaspard can be found in [17].

*4.1.2 Principles of Transformation Chains in Gaspard* For pragmatic reasons (particularly because there are few editing tools available for MARTE models), the input to each transformation chain is a UML model, expressed in a way that is compliant with the MARTE profile. This model is transformed into a MARTE model: the transformation from the profiled UML metamodel to the MARTE metamodel[5] is a classical transformation, specified in QVT-Operational (we do not go into

[4] https://gforge.inria.fr/frs/?group_id=768
[5] The version that is used is available at: https://speedy.supelec.fr/Papyrus/svn/Papyrus/extensions/MARTE/head/org.marte.metamodel/model/. The metamodel provided in the profile standard specification is

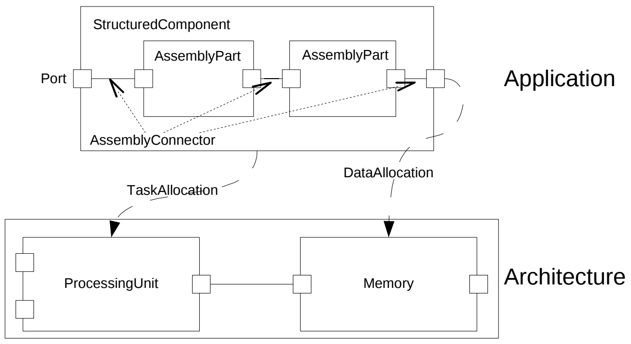


**Fig. 7** Sketch of a MARTE model

further details of this here; see [17]). In the rest of this section, we consider the MARTE metamodel to be the input language of the chains.

The generation of code from the MARTE metamodel is complex and requires numerous intermediate steps. In the initial version of Gaspard, the intermediate metamodels[6] represented specific abstraction levels (*e.g.* HDL, functional, pattern accurate), where the simulation, the synthesis or execution is performed. For example, the initial VHDL chain generates VHDL code from a UML model enhanced with the MARTE profile through two intermediaries steps: a MARTE model and a HDL model including hardware accelerator characteristics. Several steps were shared between several chains, such as the transformation from the MARTE profile to the MARTE metamodel, or the transition from using polyhedra to using loops to express the way that data are manipulated in array. The shared transformations already have a specific intention; but in concrete terms around 90% of the transformation logic exists to copy elements. Building a new chain, for example towards Verilog or OpenMP-Fortran, would be straightforward since only the corresponding code generators would need to be added; those could be reused in different context when the corresponding metamodel is reused. In order to reduce the complexity of the transformations and enhance their reuseability and maintainability, a new version of the chains based on the localised transformation has been designed.

The MARTE metamodel does not contain the essential concepts to produce code in languages such as OpenMP, OpenCL, SystemC or pThreads. One way in which to deal with this limited expressiveness is to add new concepts to a metamodel; *inout* transformations on the MARTE metamodel are not enough. Additionally, code generation for the various languages mentioned earlier *share* many characteristics, including: the mapping of data onto memory; the definition of local task graph when tasks are hierarchical; scheduling, etc. We thus decided to collect these recurring steps and define localized transformations, each dealing with one of these specific tasks. The next section presents some of the localized transformations available in our library and illustrates how three chains are built from some of these transformations.

not directly implemented; it has been slightly modified to produce this version.

[6] Some metamodels from the initial version and of the current one are available online: http://www.lifl.fr/∼etien/mmGaspard/

### *4.2 Embedded System Code Generation*

This section briefly describes some of the localized transformations available in Gaspard for generating code from a MARTE model; we do not dwell on the embedded systems details herein, but rather focus on the localized transformations and the development of a transformation chain. Some of the transformation chains we develop are illustrated in Figure 10. This represents, for example, a transformation from a MARTE model to an implementation in SystemC code.

*4.2.1 Description of transformations* The Appendix presents most of the transformations of our library and indicates their purpose. In the following paragraphs, we briefly describe some of these localized transformations.

*Port Instance Introduction.* The MARTE metamodel does not contain the port instance concept. A port instance corresponds to the port of a component instance. Since it is easier to manipulate the port instance than the couple (port, component instance), this localized transformation (*MartePortInstance*) introduces the port instance concept in the MARTE metamodel, and ensures that relevant connectors are established.

*Explicit Mapping.* An application component which is not explicitly mapped through an *Allocation* link onto a processing unit is implicitly mapped onto the same processing unit as the *StructuredComponent* which contains it. However, the code generation requires explicit mapping information for each component. This localized transformation aims to explicitly provide a mapping for each application component.

*Task graph definition and Scheduling.* Similar to the Class/Property relationship, the task hierarchy is defined bi-level through bi-level (*i.e.* a task contains parts that refers to task that at their turn may contain part and so on). However, in order to generate code, a global task graph has to be computed. This activity is divided into two parts. A localized transformation (*synchronization*) performs the task graph associated to each *StructuredComponent*. An other one (*globalSynchronization*) defines a global task graph representing the full application. From this global graph a valid scheduling is computed by a third localized transformation (*scheduling*). If we want to compute an optimized scheduling according to different criteria, only this latter localized transformation needs to be rewritten. The two other localized transformations remain unchanged.

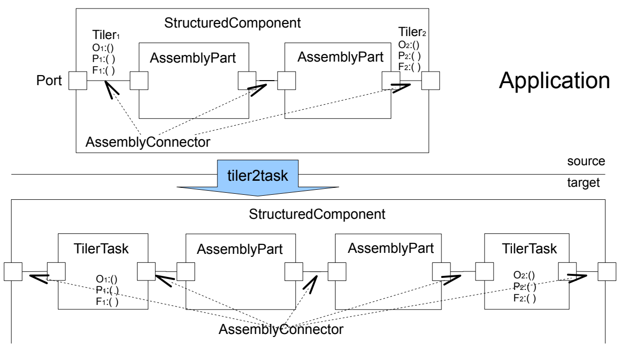


**Fig. 8** Sketch of tiler2task transformation

*4.2.2 Example of a localized transformation* In this section, we illustrate an example of a localized transformation available in Gaspard. The *tiler2task* transformation has been chosen since it gathers all the characteristics of a localized transformation and it is easily understandable, even for non-specialists in Embedded Systems. The description we provide is independent of any transformation language. Nevertheless, an excerpt of the transformation implementation is presented.

Figure 8 illustrates the principles of the transformation. In a MARTE model, a topology can be associated to a connector. There exists two type of topologies. When it is associated to a connector between a port of a *StructuredComponent* and a port of an *AssemblyPart* it is called *Tiler*. The *Origin*, the *Paving* and the *Fitting*, enables to specify how the data are consumed or produced by each task. The connectors are thus more than simple connectors. In order to generate the systems, these connectors have to be transformed into tasks and mapped onto processors. The *tiler2task* transformation creates new tasks (*TilerTask*) from connector with an associated *Tiler* topology. The *StructuredComponent* contains *AssemblyParts referring to* these newly created tasks. After this transformation, the concept of *Tiler* no longer exists.

The transformation (illustrated in Figure 8) highlights the key concepts: those that will be transformed because they no longer exist in the output, and those that are created. The metamodel associated to this localized transformation is built from these concepts. Due to the way the merge operator is usually implemented (and thus also the `Extend` operator), the hierarchy used in the metamodel to which the transformation is extended (*i.e.* MARTE in our case) has to be respected. The useful concepts can be identified in one shot or by successive iterations, as for any other transformation, except that they have first to be identified in the MARTE metamodel (in our case) and then in any metamodel associated to a localized transformation based on it. If some concepts are useful, but do not exist in any already existing metamodel, they have to be created and an intermediate localized transformation has to be defined.

Figure 9 presents the union of the input and output metamodels of the *tiler2task* localized transformation, resulting from this process. The concepts present only in the input metamodel are in the red box, whereas those only belonging to the output metamodel are in the blue box with grey background.

The input metamodel specifies that a *StructuredComponent* representing a piece of the application (*i.e.* without any *ClassifierTypeExtension*[7]) contains *Properties* that can be either an *InteractionPort* or an *AssemblyPart*. *AssemblyParts* and *StructuredComponents* are linked together by *PortConnectors* via their *AbstractPorts* (*i.e. InteractionPort* for the *StructuredComponents* and *PortPart* for the *AssemblyParts*). A *Tiler* can be associated to the *PortConnector* as a *LinkTopology*; it refers to an *IntegerVector* and two *IntegerMatrix* representing respectively the origin, the paving and the fitting.

The output metamodel is almost identical except than it contains a *TilerTask* that is a *StructuredComponent*. A *TilerTask* has a direction and refers to an *IntegerVector* and two *IntegerMatrix* representing respectively the origin, the paving and the fitting.

These metamodels are small; they each contain around 25 concepts. However, the extended metamodel used by the transformation chain that generates SystemC code contains a total of 260 concepts.

The transformation consists of 125 lines of code and is composed of 9 rules and 2 helpers:

– Rule *addTilerTaskInstance()*: add to each *StructuredComponent* of the application (*i.e.* without any *ClassifierTypeExtension*) an instance of a *TilerTask* created from each *PortConnector*.
– Rule *toTilerInstance()*: create an *AssemblyPart* whose type is the *TilerTask* creating from the *PortConnector*. The *PortParts* of this *AssemblyPart* refer to the port of the *TilerTask*. The shape is created from the shape of the ports and tasks linked by the original connector. Finally, new *PortConnectors* are created and added in the *StructuredComponent* in order to link the newly added part to the other ones.
– Rule *toTilerTask()*: create a *TilerTask* differently according that the data move from the *StructuredComponent* to the *AssemblyPart* or from the *AssemblyPart* to the *StructuredComponent*.
– Rule *toInTilerTask()*: transform a *PortConnector* with a *Tiler* typology whose direction is 'in' into a *TilerTask* with the same direction. The *source* and *target* of the *TilerTask* are the same as the ones of the original *PortConnector*. The repetition space corresponds to the shape of the *AssemblyPart* target of the original *PortConnector*. Ports are created. Fi-

[7] The *ClassifierTypeExtension* enables to extend some concepts similarly to the stereotype mechanism. It exists in the MARTE metamodel.

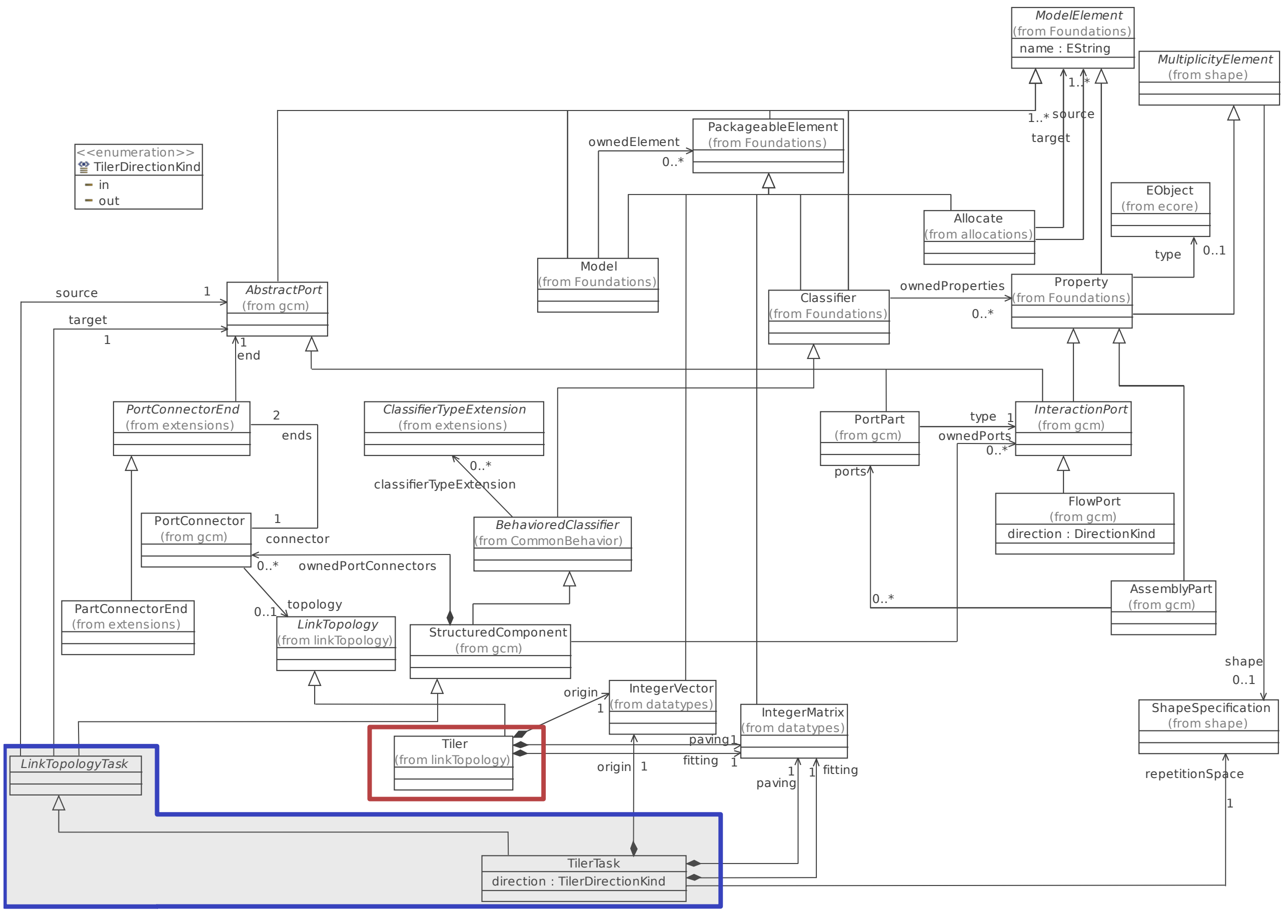

ModelElement (from Foundations) name : EString
MultiplicityElement (from shape)
PackageableElement (from Foundations)
ownedElement 0..*
<<enumeration>> TilerDirectionKind in out
Allocate (from allocations)
EObject (from ecore)
Model (from Foundations)
Classifier (from Foundations)
Property (from Foundations)
ownedProperties 0..*
AbstractPort (from gcm)
PortConnectorEnd (from extensions)
ClassifierTypeExtension (from extensions)
PortPart (from gcm)
InteractionPort (from gcm)
FlowPort (from gcm) direction : DirectionKind
PortConnector (from gcm)
BehavioredClassifier (from CommonBehavior)
PartConnectorEnd (from extensions)
LinkTopology (from linkTopology)
StructuredComponent (from gcm)
AssemblyPart (from gcm)
IntegerVector (from datatypes)
IntegerMatrix (from datatypes)
ShapeSpecification (from shape)
LinkTopologyTask
Tiler (from linkTopology)
TilerTask direction : TilerDirectionKind

**Fig. 9** Union of the input and output metamodels of the Tiler2Task localized transformation

nally, the origin, paving and fitting information takes the same value that the original *Tiler*.

– Rule *toOutTilerTask()*: is equivalent to the previous rule; only the direction and thus the *source* and *target* references are different.
– Rule *resolveTilerTask()*: add the newly created *TilerTask* in the elements collection of the model.
– helper *removePortConnectorAndTiler*: remove the *PortConnectors* from which *TilerTasks* have been created. The associated *Tiler* are also removed.
– helper *removeElt()*: effectively remove the elements from the model.

Listing 1 in Appendix B presents an excerpt of this localized transformation implemented using the QVTO language. It is defined as an *inout* transformation based on the metamodel corresponding to the union of the small input and output metamodels; it will be executed on a model conforming to the union of the small input metamodel and the MARTE metamodel[8]. It would have been possible to consider each *modeltype* as an independent metamodel, for more flexibility. However, to remain consistent with the MARTE metamodel, we considered a single metamodel as input of the *tiler2task* transformation. Moreover, since the transformation is written in *inout* mode, the removal of elements has to be explicitly expressed. After the execution of the transformation, the original *PortConnectors* will be removed from the model, others will be created to connect the *TilerTask* to the other parts of the *StructuredComponent*. In the opposite, concerning the *Tiler*, they will all be removed; no more will exist in the target model and the concept disappears from the target metamodel.

All the other transformations of the Gaspard environment are based on the same approach: associate a specific purpose to each and restrict the input metamodels to the set of concepts involved in the transformation.

*4.2.3 Description of chains* In this section, we briefly illustrate how to compose transformation chains from localized transformations. Figure 10 presents on the same diagram several transformation chains for producing OpenMP, OpenCL, pThread and SystemC code from MARTE models, respectively.

In Figure 10, four chains are designed. Each chain is composed of several localized transformations repre-

[8] The notion of port instance does not exist in Marte, it has been introduced by the MartePortInstance localized transformation in order to simplify the following treatments. The *tiler2task* localized transformation requiring the concept of *PortInstance* has to be executed after the MartePortInstance transformation.

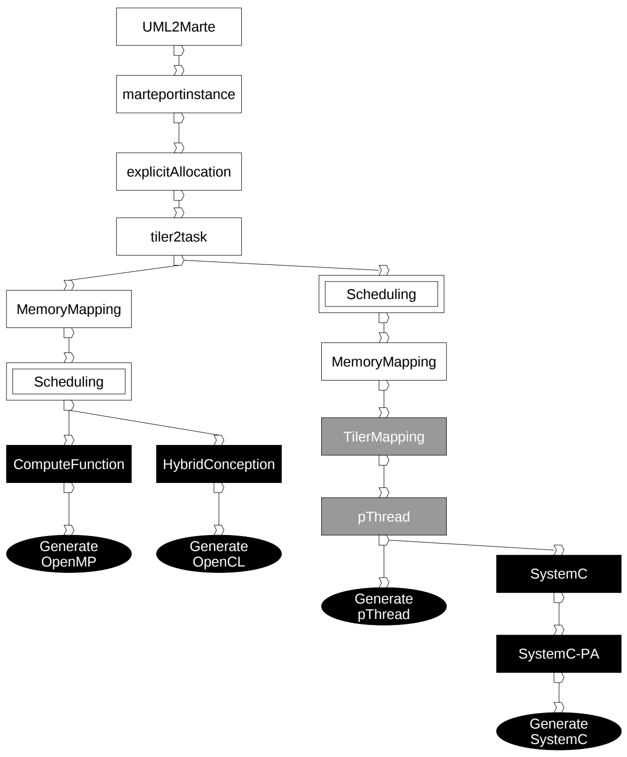


**Fig. 10** Transformation chains from a UML enhanced with the MARTE profile model to OpenMP, OpenCL, pThread or SystemC code

sented by rectangles. Each transformation has parameters typed with the metamodel associated to the transformation. Similarly, the code generations, represented by ellipses only have an input parameter. The scheduling is computed by chaining three localized transformations: (*synchronization*, *global synchronisation* and *Scheduled*). They have been gathered in a single box named *Scheduling* and represented with a double rectangle. Representing the four chains on the same diagram highlights the similarities between them but each chain is entitled to be executed separately. Before executing a chain for the first time, the localized transformations composing it have to be extended using the `Extend` operator.

Some transformations — such as the introduction of port instances (*marteportinstance*), explicit mapping (*explicitallocation*), the task graph definition and scheduling (*Scheduling*) or memory mapping (*MemoryMapping*) — are used in almost all the chains. *Scheduling* and *MemoryMapping* appear several times in the figure in order to show that they can be intertwined since they are commutative (their execution order has no effect on the models). They are the only two transformations that are commutative. Indeed, the chaining order shown in Figure 10 relies on the application of the constraints defined in section 2.4. Moreover even if based on a type analysis, the explicitAllocation transformation can be placed somewhere else in the chains, the produced systems will be different since other transformations use the *Allocation* concept and that the explicitAllocation transformation introduces some instances. Some transformations may be used by only some chains such as *pThread* that belongs to the chains towards *pThread* and towards *SystemC*. Those transformations that are available in more than one chain but not in all the chains are represented in grey. The transformations dedicated to a single chain such as *ComputeFunction* for the chain towards OpenMP are black colored. So the more specific a transformation is, the darker is the rectangle representing it. Each localized transformation is executed only once by chain but can be reused in different chains. Reusability of localized transformations between chains is very high in the Gaspard framework.

Thus, once a chain is defined, defining a new chain to target a new language is straightforward, since some of the steps used to reach the code can be shared. We have presented here four chains; the fifth, towards VHDL, requires the definition of a specific localized transformation to add to each component a clock and a reset port and another transformation to target specificities of the RTL languages. Furthermore, it does not use the *MemoryMapping* transformation. The Gaspard framework allows a user to target five different implementation languages each one produced by a chain built from localized transformations available in the library.

The transformation scenarios illustrated here are not isolated examples; there are many situations in which several related technologies can be targeted from the same input metamodel. In such cases, it is useful to be able to reuse localized transformations, as in many cases some of the same steps in the transformation chains can be shared.

## 5 Related Work

### *5.1 Decomposition of transformations*

In [19], Hemel *et al.* describe the decomposition of a code generator into small transformations that can be composed. The transformations are a set of rewriting rules. Such an approach eases the maintainability and the extensibility of the code generator. The composition of the transformations relies on annotations. Strategies enable the specification of the chaining. Furthermore, the authors introduce the distinction between local-to-local and local-to-global transformations according to the impact local or global of the transformations on the models. Similarly, in [38] Vanhooff *et al.* highlights the benefits of *breaking up large monolithic transformations into smaller units that are more easily definable, reusable, adaptable, etc.* Their decomposition approach relies on the identification of the different functionalities and the analysis of their dependencies. They, for example, prune the initial transformation with the functionalities that do not correspond to the core issue of the initial transformation or those that can be useful to other transformation units. By analogy with component interfaces, the authors stress the importance of dependencies between transformations. If these approaches have the same objectives than ours, no information is given concerning the "localized" character of the transformations. The example of these papers only concerns refactorings (*i.e. inout* transformation). Our approach goes further by generalizing the *inout* mechanism to transformations where concepts are added or suppressed in the output metamodel.

In [24], Oldevik provides a framework to build composite transformations from reusable transformations. The author assumes that a library of existing transformation is readily available. The granularity/locality degree of the transformations is not specified. In the example presented by the author, the transformations seem to be relatively complex and not focus on a specific functionality such as in [38] or in our approach.

In [25], Olsen *et al.* define *a reusable transformation [as] a transformation that can be used in several contexts to produce a required asset.* In practice, the smaller transformations are, the more they are reusable. Furthermore, they identify several techniques allowing the reuse of transformations such as specialization, parametrization or chaining. Nevertheless, no indication on the characteristics of the transformations or on the way to practically and concretely reuse transformations are provided.

In [31], Sànchez and Garcia argue that model transformation facilities are currently too focused on rules and patterns and should be tackled at a coarser-grained level. To make model transformation reusable as a whole, authors propose the *factorization* and *composition* techniques. The *factorization* technique aims at extracting a common part of two existing transformations in order to define a new transformation. The *composition* composes two transformations to create a new one. Those techniques have a major drawback. They require a connection between the input and the output metamodels of the transformations (their intersections must not be empty). The authors try to get around this issue using the notion of model type [35] and using phase mechanism. This mechanism provides operators to compose transformation definitions by means of the trace information. Our approach would overcome this drawback because it virtually increases the input and the output metamodels of existing transformations. Furthermore, the authors consider that a model can conform to several metamodels. Several namespaces are thus defined for each model. The authors propose a domain specific language (DSL) to set the models to be used in a transformation execution and to bind the namespaces with the metamodels. Finally, since the authors do not impose that the metamodels involved in the various transformations concern the same context such as MARTE in our case, they have to define bridges between the metamodels. Our work relies on the same main idea. However, we compose localized transformation. This idea is completely new, and its implementation is fully independent from any transformation language. We impose for the moment that the contexts of the localized transformation to compose have common elements but we are working on addressing this issue and improving the generality of the notion.

### *5.2 Chaining transformations*

In [28], Rivera *et al.* provide a model transformation orchestration tool to support the construction of complex model transformations from those previously defined. The transformations are expressed as UML activities. As such, they can be chained using different UML operators: composition, conditional composition, parallel composition and loop. Thus *"a chain is composed of transformations which act as processing nodes. Parameters represent the consumed and produced data by transformations. Transformations are wired together by directed connectors"*. Only heterogeneous transformations can be chained. The reuse of a transformation in different chains is thus limited due to the required inclusion of the output metamodel of the first transformation in the input metamodel of the successive one. This approach thus has restrictions in terms of reusability, adaptability and ease of construction when contrasted with localized transformations.
In [41], Wagelaar *et al.* propose the mechanism of module superimposition to compose small and reusable transformation. This mechanism allows them *to overlay several transformation definitions on top of each other and then to execute them as one transformation.* This approach is comparable to ours since it generalize the *inout* mechanism to *in to out* transformations. However, this approach relies on transformation language characteristics whereas our is fully independent.

In [22], Mens *et al.* explore the problem of structural evolution conflicts by using graph transformation and critical pair analysis. The studied transformations are refactorings (*i.e.* endogenous transformation). The modifications that a transformation can perform in such cases are precisely prescribed. Nine modifications are highlighted in the paper. With such a limited number, it is possible to study in detail when the modifications can be chained and, when doing so, if they are commutative. In the current paper, we deal with localized transformations where there are no predefined restrictions on when they can be chained. As a result, it is impossible to identify and foresee all potential cases of chaining, and thus to provide fine-grained analysis (and pre-identify all valid combinations of localized transformations) as is done in [22] using critical pair analysis. In other words, section 2.4 is similar to Mens et al.'s critical pair analysis in cases where transformations are localized. But due to the intrinsic nature of the localized transformations in comparison of refactoring, the results of [22] are coarser grain.

### *5.3 Generic transformation*

In [11], de Lara *et al.* provide mechanisms to enable the correct reuse of graph transformation systems across different metamodels. For this purpose, they use templates defined over variable types that need to be bound to type of specific metamodel, and not over the types of concrete metamodels. Not every metamodel may be considered a valid binding for the variable types used in a graph transformation template. In fact, only the metamodels satisfying a *concept* may be correctly bound. A concept enables the definition of a family of metamodels sharing some requirements. The templates defined in this approach are very small covering the concrete metamodels.

Other work on graph transformations enable reusability using parameters. For example, the VIATRA2 framework [5] supports generic rules where types can be assigned to rule parameters.

In [8], Cuadrado *et al.* adapt generic programming methodologies to model transformation in order to make them reusable. Similar to programming templates, they build generic model transformation templates, *i.e.* transformation in which the source or the target domain contains variable types. The requirements for the variable types (needed properties, associations, etc.) are specified through a concept. Concepts and concrete metamodels are bound in order to automatically instantiate a concrete transformation from the template. The resulting transformation can be executed as any other transformation on regular instances of the bound metamodels. Since the concept can be bound to several metamodels, the generic transformation template can be reused several times. This approach has a major drawback, the concept must cover the whole metamodel to which it is bound. In fact, the authors do not specify this limitation, but they do not explain what happens in case where the generic concept only covers a subpart of the specific metamodel. And yet, such cases occur frequently since it seems unbelievable that a concept fully covers all the metamodels for which we want to adapt the generic transformation template. Otherwise, the generic transformation template is not really reusable.

In [34], Sen *et al.* propose to define reusable transformations with generic metamodels. The actual transformations result from an adaptation of a generic transformation using an aspect based approach. A model typing relationship binds the elements of the generic metamodel and those of the specific metamodels. This approach has a drawback similar to Cuadrado's approach. Moreover, the input metamodel must only cover the transformation; the author suggest pruning the input metamodels in order to satisfy this constraint.

We thus believe that these approaches could benefit from the localized transformation notion and its associated implicit copy function in order to permit a partial cover of the concept on the specific metamodels with which they are bound. As explained in conclusion, developing generic localized transformations is part of our future works.

### *5.4 Managing the copy*

In graph transformations, it is often necessary to copy, delete or move entire subgraphs that match a specified subgraph. Thus, in [4], Balasubramanian *et al.* propose to simplify model transformations by allowing the selection of subgraphs and the performance of a delete, move, or copy operation in a context of a rule in a transformation. For this purpose, the notion of *Group* has been introduced and the copy, move or delete operator is applied by iterating on the elements of the *Group.*

In [30], Rose *et al.* introduce the notion of *conservative copying* in the context of model migration. Metamodel changes can affect conformance. *Model migration is a development activity in which models are updated in response to metamodel evolution to re-establish conformance. Conservative copy is an algorithm that copies model elements from original to migrated model only when those model elements conform to the evolved metamodel.* The other elements are migrated according well defined policy. This algorithm allows engineers to focus only on elements impacted by the metamodel evolution. It has been implemented in Flock, a module of the Epsilon platform [13]. The implementation of our work relies on the same idea: avoiding to the designer to write transformation rules corresponding to the copy of elements.

A similar mechanism, *refining mode*, is used in ATL. It has been introduced because this language intrinsically only supports exogenous transformation. Indeed, *by default, ATL does not transform anything in the input models and will simply give back an empty output model. For refinement or refactoring transformations, most elements should simply be copied and only a few elements are modified. In ATL, this means that every refinement/ refactoring transformation consists mostly of copying rules. ATL refining mode has been introduced to tackle this issue* [40]. The refining mode can only be used for endogeneus transformations, *i.e.* when source and target model share the same metamodel [36].

We propose a mechanism that copies the whole model and considers it an instance of a metamodel including the input and the output concepts. Moreover, in Rose's approach the distinction between the concepts impacted by the evolution and the others is already done. Such a verification can take time if the model is large. Our mechanism seems more efficient with very large models and moreover if few concepts are impacted by the evolution of the metamodel. Conservative copy and implicit copy are thus very similar and may eventually be interchangeable. However, they are each part of a larger process, model migration and transformation chain using localized transformation respectively. The reminder of the approaches is fully different since they aim at different target.

## 6 Discussion

*Construction, Decomposition.* Localized transformations address concerns about the monolithic nature of many transformation chains. They promote modularity and reusability. However, using such transformations implies that engineers should focus on the development of transformations and their chaining more, with less focus on the development of metamodels. Finding the most appropriate size of a component is a recurrent issue in the component based approaches. The decomposition mostly depends on the designer's experience and may lead to several solutions. Furthermore, defining a chain in a reusability purpose requires an ability to abstract the process underlying the transformation chains. Such a variation in the development is exactly the same as adopting a component based approach in system development. The transformation chains for OpenMP, pThread, OpenCL and SystemC share most of their localized transformations because a component based approach has been adopted and the decomposition was pertinent.

There exists two ways to build a localized transformation: from scratch or by decomposing an existing transformation. In the first case, an intention has been identified; the concepts useful to perform the transformation are selected, and the first versions of the metamodels are designed. As for any other transformation, the metamodels or the localized transformations can be adapted through several cycles. In order to be standalone, the metamodels associated to the localized transformation must contain all the elements, references and properties manipulated (*i.e.* modified, created, removed or accessed) by the transformation. In order to apply the `Extend` operator, the hierarchy of concepts should be the same in the various metamodels and thus follow the one of the domain (MARTE in our case study and UML in the motivating example). In the second case, the various parts of the existing transformation have to be identified. The corresponding metamodels have to be built by gathering all the elements, references and properties manipulated by the transformation. The chaining constraints enable to define a chaining order.

*Time saving.* Building the first chain (when no preexisting localized transformations are available) takes approximately the same amount of effort as building a non-localized transformation. Indeed, in traditional approaches, the intermediate metamodels have to be defined, the transformations between them written and validated and the resulting system has to be tested. The involved metamodels are often large, leading to transformations that can be difficult to specify and to test. In our approach, the number of metamodels and transformations is greater but the complexity to specify, test and validate each of them is reduced.

Variations may result from the size of the input metamodel. However, the time saved for the development of the second and following chains is around 1 to 4. Indeed, only the transformations dedicated to the new target and the code generation have to been developed. Such an improvement does not generally exist for traditional approaches since transformations are not easily reusable.

Figure 11 shows the development time for some chains of the Gaspard environment using a traditional approach, *i.e.* using out-place transformations (for the first version of the environment) and using localized transformations. Each chain corresponds to a prototype that has been developed by a different engineer, with approximately the same skills in MDE, and the same knowledge of supporting tools. So 8 engineers specialising in embedded systems and the targeted domains have each developed a chain. Furthermore for each version, they were helped by an expert in MDE. Developing chains is a non-trivial process; we have not proceeded to a scientific experiment where several people developed the same transformation chain to avoid bias. The results presented in Figure 11 are an initial indication of the effort and expense of developing such transformation chains, and should serve to give a rough indication of overall costs. The chain targetting OpenMP was the first one developed using the methodology proposed in this paper. The figures concerning the localized transformations are not biased by the fact that it was the second version. Indeed, the algorithms were already well known for the first version. No line of the first version, no metamodel have been reused in the second one.

*Reusability.* Using localized transformations is valuable in the context of developing a family of related transformations. Transformations constitute a family when they exhibit similarities and variabilities. Such a context occurs, for instance, when various technologies are targeted from the same core source language, like in our example where OpenMP, OpenCL, pThread and SystemC code are generated from the MARTE metamodel, or in information system design if the J2EE technology or .NET technology are each being targeted. It is also possible to

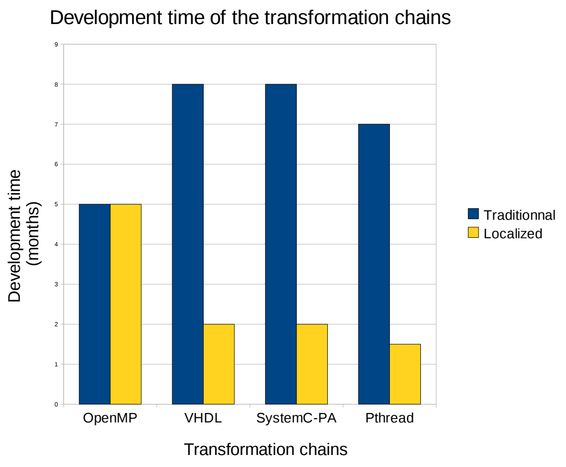


**Fig. 11** Development time for traditional approach and using localized transformations

target the same technology, but various frameworks as for example in the context of information systems where the Java technology is targeted but using the JSF, JPA or Spring framework.

Figure 10 shows that among the 13 localized transformations identified in the Appendix, four (*i.e.* 30.7%) are used within a single chain. These four transformations are the last ones in their chains, respectively, and thus manipulate concepts near to the target language. Seven localized transformations (53.8%) are used in at least 4 chains and two localized transformations (15.5%) are present in two chains. It clearly appears that the more specific the transformations are, the less they are present in various chains.

Table 1 presents another aspect of Figure 10. It highlights the percentage of reused vs specific transformations used in a chain. Each line gathers for each chain (labelled with the name of the targeted language) the total number of transformations, the number of them that are reused in at least one other chain and the associated percentage between brackets as well as the number of specific transformations in the chain and the corresponding percentage. The total number includes the localized transformations but also the *UML2Marte* transformation and the code generations. Despite taking into account all the transformations in the computation of the percentage, the rate of reused transformations per chain is pretty high, between 76.92 and 90.90%. It has to be noticed that these numbers could be even higher by taking into account only the localized transformation. Indeed, each chain targets a different technology and its associated language. Thus, each chain has its own code generation that is counted as a specific transformation. Similar results are observed in the context of information system where different frameworks using the Java technology, (*e.g.* JSF, JPA, Spring…) are targeted. The specific transformations are no more the one closest to the technology just before the code generation, but

**Table 1** Percentage of reused/specific transformations per chain

| Chain name | Total number | Reused | Specific |
|---|---|---|---|
| OpenMP | 10 | 8 (80%) | 2 (20%) |
| OpenCL | 10 | 8 (80%) | 2 (20%) |
| pThread | 11 | 10 (90.90%) | 1 (9,10%) |
| SystemC | 13 | 10 (76.92%) | 3 (23.08%) |

those taking into account the specificities of the framework. The code generation is shared between different chains. Once again these results are promising insights, as they indicate the reusability of the chains is very high within a domain when using localized transformations. However, reusability is reduced between chains of different domains. The introduction of genericity in localized transformations is likely to enhance reusability.

In a situation where a unique transformation chain or even a unique transformation is designed, the benefit of localized transformations can be more debatable. Indeed, Figure 11 shows that if no localized transformation is available off the shelf, the time saved is null. On the one hand, the number of transformations available will grow; conversely if the input metamodel is very large, localized transformations enable a separation of concerns and thus improve the design, the understandability and maintenance.

*Test.* With our approach, testing or validating model transformations can require less effort, because generally each building block of the model transformation is smaller. Indeed, each localized transformation has a unique and very specific purpose. By comparison to large transformations, it is thus easier to check that the transformation does what is expected or not. Furthermore, thanks to the localized characteristic of our transformations, it should be possible to perform a test fully covering the metamodels. Indeed, one of the crucial issue in test activity is to qualify the input data *i.e.* the ability of the data set to highlight errors in a program or a transformation. In [15], Fleurey *et al.* propose to qualify a data set according to the covering of the input domain. Sen *et al.* enhance this approach by pruning the input metamodel beforehand [33]. This activity is not useful since only the involved concepts appear in the metamodels associated to a localized transformation. Otherwise, traditional approaches to test transformations such as [2], [16], [3] can be applied on localized transformation.

However, the number of transformations in a chain drastically increases. More transformations have to be tested and their chaining also. Testing the chaining relies on typing or structural constraints (*e.g.* can these two transformations be chained and in which order?) and on business or semantic constraints (*e.g.* are all the requirements fulfilled by the chain with no overlap?). Using a rigorous traceability mechanism enables the designer to follow along the chain what becomes an input element or how an element of an intermediate model has been created. Thanks to UIDs, these mechanisms are also compatible with localized transformations [3]. Precise chaining constraints have also to be specified.

*Performance.* The execution of a localized transformation is composed of two parts, the implicit copy and the transformation it self. Concretely, the implicit copy is a copy of the file corresponding to the input model. Thus, the input model may contain few elements or several hundreds, the copy is almost instantaneous. Using localized transformations does not reduce the performance; the execution time of the chains in their initial version or using localized transformations is approximately the same.

*Modularity and Understandability.* In [37], the authors consider that the modularity of a model transformation positively depends on the number of modules it contains and negatively depends on the unbalance (*i.e* module size compared to the average of module size) and the number of main functions per module. By analogy, we could consider that the modularity of a transformation chain positively depends on the number of transformations and negatively depends on the unbalance and the number of rules or queries by transformation. The chains of the Gaspard environment count each between 10 to 12 transformations. By opposition, in the primary version of Gaspard that used traditional chain approach, the chains counted 3 to 5 transformations, without almost any reuse between chains. Each localized transformation contains in average 15 rules or queries. Some as the *pThread* localized transformation counts much more rules or queries (26) others as the scheduling transformation counts far less (4). Furthermore, even if it is hardly measurable and a little bit subjective, we advocate to focus each localized transformation on a single purpose. These figures and this recommendation illustrate the increasing of modularity when using localized transformations in chains.

In [37], the authors write: *"A large number of modules is no guarantee for an understandable model transformation. The modules should be balanced in terms of size and functionality."*. Similarly, we can affirm that a large number of transformations is no guarantee for an understandable transformation chain. However, localized transformations are by essence very small (mostly less than 150 lines of code and in average 15 rules or queries), focus on a single intention and work with very small metamodels (around 25 metaclasses by metamodels used in the Gaspard environment) whereas the original input metamodel is large with 260 concepts. Thus, each localized transformation is more easily understood than a traditional transformations and by transitivity also the chain it self. In fact, the complexity is transferred to the composition of the chain in order to ensure that the localized transformations can be chained and all together fulfill the specifications.

*Chaining localized transformations.* In traditional approaches, the order of the transformations is imposed by the metamodels on which they rely. Indeed, the input metamodel of one transformation should be the output of the preceding one. The localized transformations introduce more flexibility that has to be managed when building the chain. In [14], we identified some chaining rules based on the inclusion of the metamodels of the localized transformation in the extended metamodels. We also highlighted that if, according to the metamodel inclusion, some transformations can be inter-twinned the produced models can be different. A verification of constraints has to be performed. These constraints concern either typing like in the chaining rule or functionality and business. We are currently working on the definition of a new abstraction level that will provide more information on the transformation while being independent of the used language, and not inspecting the content of the rules. This new language will allow a finer-grained analysis relative to the typing constraints.

Today the tool supports the chaining rules presented in [14] and will later integrate the other constraints. However, if the actual version enables the chain designer to automatically check the chaining constraints, and thus eliminates many cases of incorrect chaining, we know that some cases required to inspect the content of the rules to validate the order.

Furthermore, the designer that will build the chain from the transformations off the shelf needs to have a great and abstract vision of the full process in order to order the transformations and identify those missing.

*Modifiability and Evolution.* By nature, the localized transformations are small and focus on a unique purpose. The metamodels involved are very small compared to the initial input metamodel. An evolution in this initial input metamodel may have different impact on the transformations. The idea is not to discuss here on the co-evolution of the metamodel and the transformation as in [20], but to suggest some evolution scenarios and to show how they can be managed in the case of localized transformations.

**Scenario 1: Modification concerning the Memory concept in the MARTE metamodel**. In the current version of the MARTE metamodel, a memory is a *StructuredComponent* with a reference *classifierTypeExtension* to a *HW_Memory*, similarly to a stereotype. Considering an evolution leading to the removal of the *classifierTypeExtension* reference and the introduction of an inheritance link between *StructuredComponent* and *HW_Memory*, as it is more usual in a metamodel. In the existing transformations used in Gaspard, only the first one *UML2Marte*, the localized transformation *MemoryMapping* and the code generations deal with the memory concept. None of the other transformations would be changed. The *UML2Marte* transformation counts around 1500 lines. Identifying where the memory concept is manipulated can be fastidious, even if in this transformation, it is only once since it establishes the bridge between the UML and the metamodel world. In the *MemoryMapping* transformation, the memory concept is everywhere. The transformation is thus very sensitive to this evolution that can lead to the rewriting of at least 60 to 70% of the transformation.

**Scenario 2: Modification concerning the Processor concept in the MARTE metamodel**. This modification is very similar to the previous one. In the current version of the MARTE metamodel a processor is a *StructuredComponent* with a reference *classifierTypeExtension* to a *HW_Processor*. In the next version, the *HW_Processor* inherits from the *StructuredComponent* metaclass. Almost all the transformations are impacted since the concept is central. Some of them as the *explicitAllocation* or *tiler2task* transformations can be very punctually impacted less than 5%. Others like *GlobalGraph* or *Scheduling* would be more impacted until 20 to 30% but none would be almost completely rewritten as in the previous scenario.

**Scenario 3: Enhancing of the scheduling**. In this scenario, no metamodel evolves. Only a transformation evolution is considered. The idea is here to enhance the way the task scheduling is currently calculated in order to optimize the generated code execution. This evolution concerns only the *Scheduling* transformation that should be mostly rewritten. In fact, this localized transformation should evolve except if an other localized transformation performing an optimized scheduling exist. In that later case, any transformation is modified, only the chain evolves to use the enhanced scheduling. Even if the optimized scheduling transformation does not exist, the evolution is very easily located, one transformation counting 50 lines of code.

Whereas scenario 1 fully benefits from the implicit copy since the modified concept is manipulated in few transformations, it is not the case of scenario 2. Indeed, the evolution concerns a concept almost used by each transformation. Each modification in the transformations is very small, but they are numerous due to the high number of transformations. More precisely, the number of transformations that are impacted differs with the number of impacted metamodels. If in the current version several metamodels adopt the same approach to design the memory (resp. the processor) concept, it could be relevant in the new version to transmit the evolution on the MARTE metamodel on the following one. The transmission of the evolution is natural with the localized transformations. Forcing adoption of the same hierarchy in the small metamodel as in the input metamodel of the chain (*i.e.* MARTE in our case) is a technical constraint linked to the way model merge operators are implemented. Such a practice likely increases the number of concepts in these small metamodels, and thus potentially the number of evolutions they can be submitted to. Furthermore, by copying the hierarchy in all metamodels where the concepts are used, implies the necessity to evolve several metamodels in the same way. Enhancement and simplification of these intermediate metamodels in order, for example, to automatically recover the hierarchy are foreseen in order to face the maintability issues partially transferred from the transformations to the metamodels.

Using "traditional" transformations, scenarios 1 and 2 are similar. Except the *UML2Marte* transformation, at least one or more transformations would be impacted. These impacts can occur anywhere in the transformations (exactly as in scenario 2 in case of localized transformations). Concerning the scenario 3, in a traditional approach, it won't be possible to interchange a transformation by another one, because each transformation does not focus on a single intention but enable to reach a specific abstraction level. It would be necessary to modify a transformation. Identifying all the rules impacted in the transformation is not an easy task and require a good knowledge of the transformation.

To conclude on modifiability, it appears that, due to its intrinsic characteristics each localized transformation is more easily modifiable. However, according to the modification several transformations or metamodels may have to evolve, since they depend on the initial input metamodel.

*Dependency with the initial input metamodel.* The running example in the context of information system design and the case study related to embedded system co-design have shown that localized transformations can be used to develop chains in different business domains. As a result, we claim that our approach is context independent. However, a localized transformation is integrated in a transformation chain only if its input metamodel is included in the extended metamodel of the previous transformation. Such a chaining condition involves a dependency of the localized transformations with the initial input metamodel. Thus, for example, in the *tiler2task* transformation detailed in section 4.2.2, the metamodels include lots of concepts from the MARTE metamodel. This dependency exists mostly with the initial input metamodel since the intermediate ones are not known and change according the transformation chaining. Other dependencies may occur, if some localized transformations require others.

In fact, the localized transformation concept is a first indispensable step towards generic transformation; the nature of the `Extend` operator and the use of transformations that are extended from operating on small input metamodels, to much larger ones, is a form of generalization. Indeed, traditionally model-to-model transformations create a new model and all the input elements that are not transformed do not exist in the output model.

Moreover, the metamodels involved in the transformations are often wide and complex - such as UML 2.x and profiles. By contrast, to be reused and generic, the transformations have to involve a few set of elements and thus small metamodels; a generic metamodel counting several dozens of elements will be hardly bound to several specific metamodels. In [23] the authors introduced the genericity to refactoring; in that case, only the concepts involved by the transformation are modified, the other remain unchanged and the input metamodel of the refactoring transformation may be very small.

The *tiler2task* transformation would be generic, if its input metamodel only contains Component, Part, Port and Connector. In that case, each of these concepts would require to be bound with a concept of the specific metamodel. In the near future, we would like to explore this opportunity, in order for our tool to support more generic localized transformations. Thus we would be able to define localized transformation that could be more largely used than only in the context of one single input metamodel.

## 7 Conclusion

In this paper we have presented an approach to manage transformation complexity. We proposed to decompose complex transformations into smaller ones, with each having conceptual integrity and restricted scope. As a result, we have introduced the concept of localized transformation. The only specificity of the metamodels associated with these transformations is that they contain only the concepts useful to the transformation and thus generally contain a few of them. The notion of localized transformations introduced in this paper relies both on the `Extend` operator, defined to extend transformations in order to chain them, and on chaining constraints. Thus, in order to satisfy typing constraints, the localized transformations need to be combined with an implicit copy/identity function in order to take into account each concept. Consequently, a localized transformation corresponds to an *inout* transformation, where some concepts may be introduced or removed (and, as a result, input and output metamodels may be different). This approach has been validated by an implementation in Eclipse, via the LTDesigner tool, provided in the Gaspard environment.

We have illustrated our approach in the context of Gaspard by describing four chains that target pThread, OpenMP, OpenCL and SystemC code respectively. These chains are built from localized transformations available in a library. Some transformations are common to all the chains whereas others are only used in one. Other contexts, such as the model-based design of product lines or environments targeting various technologies or languages may also benefit localized transformations in order to enhance reusability, modularity and modifiability. A motivating example from the context of information system design supports our argument that our approach is independent from the application context.

Our future work is twofold: providing finer chaining constraints differentiating concepts for which instances are read, modified, created or removed by the transformation enactment; and exploring the notion of generic model transformations, as described in the previous section. In particular, we are looking to couple notions of genericity in template-based programming with the notion of localized transformation developed in this paper.

## References

1. M. Alanen and I. Porres. A metamodeling language supporting subset and union properties. *Software and System Modeling*, 7(1):103–124, 2008.
2. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Traceability for mutation analysis in model transformation. In *Proceedings of the 2010 international conference on Models in software engineering*, 2011.
3. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Using trace to situate errors in model transformations. In J. Cordeiro, A. Ranchordas, and B. Shishkov, editors, *Software and Data Technologies*, volume 50 of *Communications in Computer and Information Science*, pages 137–149. Springer Berlin Heidelberg, 2011.
4. D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, and G. Karsai. A subgraph operator for graph transformation languages. *ECEASST*, 6, 2007.
5. A. Balogh and D. Varró. Advanced model transformation language constructs in the viatra2 framework. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1280–1287, New York, NY, USA, 2006. ACM.
6. J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *IEEE Computer*, 40(6):27–35, 2007.
7. J. Cordy. Eating our own dog food: DSLs for generative and transformational engineering. In *GPCE*, 2009.
8. J. Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: *Write Once, Reuse Everywhere*. In J. Cabot and E. Visser, editors, *ICMT, International Conference on Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2011.
9. K. Czarnecki and S. Helsen. Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.
10. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
11. J. de Lara and E. Guerra. Reusable graph transformation templates. In *AGTIVE*, volume 7233 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2012.
12. M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC*, pages 53–62, 2008.

13. Epsilon. `http://www.eclipse.org/epsilon/`, 2013.
14. A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining independent model transformations. In *Proceedings of the ACM SAC, Software Engineering Track*, pages pp. 2239–2345, 2010.
15. F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon. Qualifying input test data for model transformations. *Software and System Modeling*, 2009.
16. F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Model, Design and Validation Workshop*, nov. 2004.
17. A. Gamatié, S. Le Beux, É. Piel, A. Etien, R. Ben Atitallah, P. Marquet, and J. Dekeyser. A model driven design framework for massively parallel embedded systems. *ACM Transactions in Embedded Computing Systems (TECS)*, 2011. To Appear. Available as the RR-6614 research report entitled "A Model Driven Design Framework for High Performance Embedded Systems" http://hal.inria.fr/inria-00311115/en/.
18. E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. transML: A family of languages to model model transformations. In *MoDELS/UML*, 2010.
19. Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 183–198, Berlin, Heidelberg, 2008. Springer-Verlag.
20. D. Mendez, A. Etien, A. Muller, and R. Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Wokshop*, Olso, Norway, Oct. 2010.
21. T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development*, 2005.
22. T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, Apr. 2005.
23. N. Moha, V. Mahé, O. Barais, and J.-M. Jézéquel. Generic model refactorings. In *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009.
24. J. Oldevik. Transformation composition modelling framework. In *Proceedings of the Distributed Applications and Interoperable Systems Conference*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer, 2005.
25. G. Olsen, J. Aagedal, and J. Oldevik. Aspects of reusable model transformations. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, pages pp. 21–26, 2006.
26. OMG. UML Profile for MARTE, Version 1.0, Nov. 2009.
27. J. Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and visualizing transformation chains. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 17–32, Berlin, Heidelberg, 2008. Springer-Verlag.
28. J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL model transformations. In *Proc. of MtATL 2009*, pages 34–46, Nantes, France, July 2009.
29. L. Rose, E. Guerra, J. Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software and Systems Modeling*, pages 1–19, 2011.
30. L. Rose, D. Kolovos, R. Paige, and F. Polack. Model migration with Epsilon Flock. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.
31. J. Sanchez Cuadrado and J. Garcia Molina. Approaches for model transformation reuse: Factorization and composition. In *Proceedings of the International Conference on Model Transformation*, volume 5063 of *LNCS*, pages pp. 168–182. Springer-Verlag, 2008.
32. D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
33. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, Oct 2009.
34. S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Software and System Modeling*, 11(1):111–125, 2012.
35. J. Steel and J.-M. Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, Dec. 2007.
36. M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Refining Models with Rule-based Model Transformations. Rapport de recherche RR-7582, INRIA, Mar. 2011.
37. M. van Amstel, C. Lange, and M. van den Brand. Metrics for Analyzing the Quality of Model Transformations. In *12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering*, 2008.
38. B. Vanhoof and Y. Berbers. Breaking up the transformation chain. In *Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005*, San Diego, California, USA, 2005.
39. B. Vanhooff, D. Ayed, and Y. Berbers. A framework for transformation chain development processes. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, pages pp. 3–8, 2006.
40. D. Wagelaar. Composition techniques for rule-based model transformation languages. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2008.
41. D. Wagelaar, R. Van Der Straeten, and D. Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 2009. Online First.
42. I. Weisemöller and A. Schürr. Formal definition of mof 2.0 metamodel components and composition. In *MoDELS*, pages 386–400, 2008.

# Appendices

## A Overview of the Gaspard Transformations

### *A.1 ComputeFunction*

**Transformation Description:** Introduce the concepts relative to procedural languages

**Main Used Concepts:** *AssemblyPart, StructuredComponent, DataAllocate, TilerTask, ShapeSpecification*

**Main Introduced Concepts:** *FunctionCall, TilerComputation, Process, Instruction, Synchro*

### *A.2 ExplicitAllocation*

**Transformation Description:** Explicits the association of each application part on the processing units, according to the association of other elements in the application hierarchy

**Main Used Concepts:** *AssemblyPart, StructuredComponent, Distribute, Allocation*

**Main Introduced Concepts:** ∅

### *A.3 GlobalSynchronization*

**Transformation Description:** Establishes a graph of tasks for the complete application from the local graphs of tasks

**Main Used Concepts:***AssemblyPart, StructuredComponent, Distribute, ClassifierTypeExtension, HW_Resource*

**Main Introduced Concepts:** *GlobalGraph, Task, TaskDependency*

**Main Removed Concepts:** *Graph, Node, IntraDependency, ExtraDependency*

### *A.4 HybridConception*

**Transformation Description:** Distinguishes the hosts and devices in case of GPU architecture and builds the functions and variables accordingly.

**Main Used Concepts:***AssemblyPart, StructuredComponent, Distribute, ClassifierTypeExtension, HW_Resource, Allocate, FlowPort, PortPart, TilerTask*

**Main Introduced Concepts:** *HybridApp, HostSide, DeviceSide, Kernel, Function, Variable*

### *A.5 Tiler2task*

**Transformation Description:** Replace the connectors between a component and an inner repeated part by a task managing the data (*TilerTask*); these connectors inevitably having a Tiler topology.

**Main Used Concepts:***AssemblyPart, StructuredComponent, PortConnector, FlowPort, PortPart, InteractionPort, Tiler, PortConnectorEnd, PartConnectorEnd, ShapeSpecification, Allocate, Distribute*

**Main Introduced Concepts:** *TilerTask, LinkTopologyTask*

**Main Removed Concepts:** *Tiler*

### *A.6 MartePortInstance*

**Transformation Description:** Introduce into the MARTE metamodel the concept of *PortInstance* corresponding to an instance of port associated to a part

**Main Used Concepts:** *AssemblyPart, StructuredComponent, FlowPort, InteractionPort, AssemblyConnector, Tiler, ShapeSpecification, Allocate*

**Main Introduced Concepts:** *PortPart, PortConnector, PortConnectorEnd, PartConnectorEnd*

### *A.7 MemoryMapping*

**Transformation Description:** Maps the data into memory: create the variables and allocate address spaces

**Main Used Concepts:** *AssemblyPart, StructuredComponent, PortConnector, PortConnectorEnd, PartConnectorEnd, PortPart, ShapeSpecification, Allocate, Distribute, HW_RAM, TilerTask*

**Main Introduced Concepts:** *MemoryMap, DataAllocate*

### *A.8 pThread*

**Transformation Description:** Transforms MARTE elementary tasks into threads and the connectors into buffers

**Main Used Concepts:** *AssemblyPart, FlowPort, ShapeSpecification, TilerTask, DataAllocate, GlobalGraph, Task, TaskDependency*

**Main Introduced Concepts:***Buffer, Thread, Function, DataAccess*

### *A.9 Scheduling*

**Transformation Description:** Schedules the tasks from the global graph

**Main Used Concepts:***GlobalGraph, Task, TaskDependency, AssemblyPart, StructuredComponent, ClassifierTypeExtension, HW_Resource*

**Main Introduced Concepts:**∅

### *A.10 Synchronisation*

**Transformation Description:** Associates, to each application component, a local graph of tasks corresponding to its parts and specify if the dependencies between tasks are spatial or temporal

**Main Used Concepts:***AssemblyPart, StructuredComponent, Distribute, ClassifierTypeExtension, HW_Resource, PortConnector, PortConnectorEnd, PartConnectorEnd, PortPart*

**Main Introduced Concepts:** *Graph, Node, IntraDependency, ExtraDependency*

### *A.11 SystemC*

**Transformation Description:** Traduce the MARTE architecture into concepts of the SystemC language

**Main Used Concepts:** *AssemblyPart, StructuredComponent, FlowPort,ClassifierTypeExtension, HW_Resource, PortPart, PortConnector, PortConnectorEnd, PartConnectorEnd*

**Main Introduced Concepts:** *Module, Socket, Process, Port, Bind, Main Channel, ScThread*

### *A.12 SystemCPA*

**Transformation Description:** Establishes the links between pThread concepts corresponding to the application part and the systemC ones corresponding to the architecture part.

**Main Used Concepts:** *Module, Main, AssemblyPart, StructuredComponent,ScThread, Function, DataAllocate, Distribute, DataAccess, FlowPort, PortPart*

**Main Introduced Concepts:** ∅

### *A.13 TilerMapping*

**Transformation Description:** Maps the tiler task onto processing units according to the mapping of the nearer tasks.

**Main Used Concepts:** *GlobalGraph, Task, TaskDependency, AssemblyPart, Distribute*

**Main Introduced Concepts:** ∅

## B Excerpt of the *tiler2task* Transformation

```
modeltype marte uses MarteLTT::coreElements::Foundations('http://fr.inria.dart.gaspard2.metamodel.marteLTT');
modeltype gcm uses MarteLTT::gcm('http://fr.inria.dart.gaspard2.metamodel.marteLTT');
modeltype rsm uses MarteLTT::rsm('http://fr.inria.dart.gaspard2.metamodel.marteLTT');
modeltype extensions uses MarteLTT::coreElements::extensions('http://fr.inria.dart.gaspard2.metamodel.marteLTT');
modeltype linktopology uses MarteLTT::LinkTopologyTask('http://fr.inria.dart.gaspard2.metamodel.marteLTT');
modeltype ecore uses ecore('http://www.eclipse.org/emf/2002/Ecore');

transformation tiler2task(inout model : marte );

main()
{
  model.objects()[gcm::StructuredComponent]-> map addTilerTaskInstance();
  model.objects()[marte::Model]->map resolveTilerTask();
  model.objects()[marte::Model]-> removePortConnectorAndTiler();
}

mapping inout gcm::StructuredComponent::addTilerTaskInstance()
  when {self.classifierTypeExtensions->isEmpty()}
   {
    self.ownedProperties+= self.ownedPortConnectors->map toTilerInstance();

}

mapping gcm::PortConnector::toTilerInstance() : AssemblyPart
  {
    name := 'tilerTaskInstance';
    type := self.map toTilerTask().oclAsType(ecore::EObject);
    ports += self.map toTilerTask().ownedPorts->map createPortInstances();
  end {
    result.map computeShape();
    self.container()[gcm::StructuredComponent]->map createPortConnectors(self, result);   }
  }

mapping gcm::PortConnector::toTilerTask() : TilerTask
  disjuncts  gcm::PortConnector::toInTilerTask,
    gcm::PortConnector::toOutTilerTask
  {
  }

mapping gcm::PortConnector::toInTilerTask() : TilerTask
  when { self.topology.oclIsTypeOf(rsm::linkTopology::Tiler) and self.getTilerDirection() = linktopology::TilerDirectionKind::_in}
   {
    name := 'tilerTaskComponentIn';
    source := self.ends[extensions::PortConnectorEnd]._end->selectOne(oclIsKindOf(FlowPort));
    target := self.ends[extensions::PortConnectorEnd]._end->selectOne(oclIsKindOf(PortPart));
    direction := linktopology::TilerDirectionKind::_in;
    repetitionSpace := self.ends[extensions::PortConnectorEnd]._end->selectOne(oclIsKindOf(PortPart)).container()![AssemblyPart].shape;
    ownedPorts += self.ends->map createTilerPort(linktopology::TilerDirectionKind::_in);
    origin = self.topology.oclAsType(rsm::linkTopology::Tiler).origin;
    paving = self.topology.oclAsType(rsm::linkTopology::Tiler).paving;
    fitting = self.topology.oclAsType(rsm::linkTopology::Tiler).fitting;
}

mapping gcm::PortConnector::toOutTilerTask() : TilerTask
  when { self.topology.oclIsTypeOf(rsm::linkTopology::Tiler) and self.getTilerDirection() = linktopology::TilerDirectionKind::_out}
   {
   ...
}

mapping inout Model::resolveTilerTask()
```

```
{
 self.ownedElement += model.objects()[gcm::
      PortConnector]->select(e | e.container()[
      StructuredComponent].classifierTypeExtensions
      ->isEmpty() and e.topology.oclIsTypeOf(rsm::
      linkTopology::Tiler)) -> map toTilerTask();
}

helper Model::removePortConnectorAndTiler() {
 var portConnectors := model.objects()[gcm::
      PortConnector]->select(e | e.container()[
      StructuredComponent].classifierTypeExtensions
      ->isEmpty() and e.topology.oclIsTypeOf(rsm::
      linkTopology::Tiler));
 var tilers := portConnectors.topology[rsm::
      linkTopology::Tiler];
 portConnectors.oclAsType(EObject)->removeElt();
 tilers.oclAsType(EObject)->removeElt();
 return;
}

helper ecore::EObject::removeElt() {
 model.removeElement(self);
}
}
```