

# Package Unmerge

Frédéric Fondement, Brice Wittmann, Pierre-Alain Muller

Université de Haute Alsace,  
12, rue des Frères Lumière  
F-68093 Mulhouse cedex - France  
{frederic.fondement,brice.wittmann,pierre-alain.muller}@uha.fr

**Abstract.** As many other software engineering activities, building metamodels is concerned with reuse. New metamodels are seldom defined from scratch, and are usually more or less similar to existing metamodels. While mechanisms, such as profile and package merge, have been defined for handling extension of metamodels, there is no such mechanism for controlling the reduction of metamodels. As a consequence, metamodels tend to contain unnecessary modeling elements, which leads to harmful overheads of complexity for both tool builders and users. In this paper, we first introduce package unmerge - a new relation between packages - which is defined as a counterpart to package merge, and next show how these two relations may be used together to fine-tune metamodel reuse.

## 1 Introduction

Building metamodels, as many other software engineering activities, can greatly benefit from reuse. This was acknowledged by the OMG, as they introduced the package merge mechanism, by which the UML language could be constructed by the fusion of existing components.

Indeed, new metamodels are very often more or less similar to existing metamodels, and being able to express these similarities may greatly alleviate the process of metamodel editing.

Profiles and package merge relations are the dominant techniques for managing extensions of metamodels. Profiles offer means for controlling how features are added to existing modeling elements, while package merge relations provides both structuring and merging policies for controlling the fusion of modeling elements.

While extension has deserved significant interest, reduction has not yet gained the same exposure. However, as pointed out in the literature [1], there are significant differences between metamodels and their effectively used subsets. In other words, metamodels contain too many features, one reason for that being that it is currently impracticable to identify and remove unneeded parts. Unfortunately, these undesirable features lead to harmful overheads of complexity for both tool builders and users.

In this paper, we examine how extension and reduction of metamodels could be expressed in a seamless way, basically by providing operational semantics for a package unmerge mechanism, built as a counterpart of the package merge.

Besides metamodeling, this work is related to several other fields, such as grammars, databases, domain specific languages and ontologies; which have all addressed this issue of merging or aligning descriptions [2, 3]. Interestingly, as for metamodels, the issue of reduction has got much less coverage than extension.

The paper is organized as follows: after this introduction, section 2 motivates the need for reduction mechanisms for metamodels, section 3 presents three dominant techniques for metamodel extension, section 4 presents our proposal for reduction (that we call package unmerge relation), section 5 illustrates the use of package unmerge, section 6 provides more in depth examples, section 7 compares our approach to others, and section 8 concludes and presents future directions.

## 2 Motivation for reducing metamodels

Over the past years, as our team was involved in different projects about meta-modeling, we have had to face a recurring issue for reusing existing metamodels: being able to simultaneously extend and reduce metamodels. While established mechanisms are available for managing extensions (such as profiles and package merge relations) we are not aware of equivalent mechanisms for reducing metamodels. This is unfortunate, because undesirable language features lead to harmful overheads of complexity for both tool builders and users.

We ran again in this situation as we were developing elements of a test generator for SysML, in the context of the VETESS project<sup>1</sup>. On one hand we had SysML models, on the other hand we had a test generator whose input language was a subset of UML. SysML is defined as a UML profile, which means that SysML models are UML models (as profiles do only decorate metamodels), and so one would naively expect an UML tool to be able to process such UML profiled model. Unfortunately, the test generator was not using UML, but a subset of UML [4]; moreover, this subset was not explicitly represented by a metamodel, but was merely implemented in the tool.

To determine the subset of UML used by the tool generator, we first developed a “footprint” system, at the EMF repository level, which earmarked all the UML constructs that the tool was manipulating as it was processing models, until we had a stable footprint of the tool (at least stable for all the models that we had at hand). We determined that two types of reductions would have to be applied to UML in order to specify the subset used by the tool: coarse grained reductions (e.g. removing use case diagrams, timing diagrams...) and fine grained reductions (e.g. generalization relations, String attributes...).

While we tried to use the modularization technique of UML, looking for those package not-to-merge, we realized that we would be able to address coarse grained reductions, but not fine grained, such as removing generalization relations which are defined in the same package than classes and associations. In short, even modular lan-

---

<sup>1</sup>. <http://lifc.univ-fcomte.fr/vetess/>

guages are not necessarily as modular as required, and reducing a language by merging only a subset of the modules is not applicable in a general fashion.

As a conclusion, another mechanism is required to extract parts of existing modules. Preferably, because footprint detection is only a workaround, we would like to be able to describe explicitly those parts to be reduced, using metamodels as for package merge relations.

### 3 Extending metamodels

In model driven realm, models are expressed in languages themselves described by other models that play the role of so-called metamodels. Hence, a given metamodel defines the set of all models that are said to conform to that given metamodel. As a consequence, enlarging the range of possible models, implies extending the corresponding metamodels. Typical mechanisms for controlling metamodel extension include UML profiles, package merge relations, and aspect weaving.

Profiles [5 - section 18] became popular as UML promoted them as a lightweight approach for language extension. Profiles define extension points (called stereotypes) for the metaclasses of a (MOF [6]) metamodel. Stereotypes can insert additional properties or constraints to the metaclass they extend. Stereotypes work as decorations, do not modify the decorated metamodels, and can be removed or swapped at any moment in the lifecycle of a model. Therefore, models remain conforming to their original metamodels (regardless of profiles).

Package merge relations [7 - section 11.9.3], as opposed to profiles, are considered heavyweight an extension mechanism, since they impact directly the metamodel elements. Package merge relations are available both in the UML standard and in the MOF metalanguage. Package merge relations combine the contents of two packages into a single one, following a recursive union-like copy approach. In case of name conflicts, conflicting elements are merged together into the same element in the resulting package. Package merge relations make the core of the modularization technique of the UML metamodel. An illustrating example is the definition of UML compliance levels. Compliance levels define the modeling concepts that must be supported by tools. A tool with compliance level L1 must support class diagrams and use case diagrams, while L2 compliance level also requires to support profiles. Since UML modeling elements are distributed across a set of packages in the UML metamodel, the L1 compliance level is formalized by a package that is merely built by merging those packages that define the necessary concepts for class and use case diagrams. Similarly, L2 compliance level is also defined by a package that merges L1 package and the package that formalizes the profile concepts (among others).

Aspect weaving was originally proposed in the context of programming [8]. Generally speaking, aspects define extension points (often called join points) where elements (often called advices) may be injected (woven in aspect-oriented terminology). Join points are conveniently specified by pointcuts, which can target different join points using a single pattern. More recently, aspect weaving has been used to alter models, and by extension metamodels [9]. Many different formalisms have been studied so

far, including UML class diagrams [10]. As MOF is also based on class diagrams, MOF metamodels may also be woven with aspect models in order to be extended.

To summarize, profiles provide a lightweight approach, that makes some metamodeling capabilities available at modeling time. Package merge relations focus on meta-modeling time. Aspect weaving, is used at modeling time, but can be used at meta-modeling time as well, since any metamodel is also a model.

## 4 Unmerging metamodels

A metamodel may be seen as a hierarchical set of information about the structure of conforming models. For metalanguages such as MOF and Ecore, packages contain classes that bear properties, and the hierarchy is represented by composition links between instances of the classes (actually, meta-classes). By altering those classes and relations, it is possible to restrict the range of conforming models. Typical modifications include removing class properties and strengthening constraints such as multiplicities. Practically speaking, this means pruning metamodel just like a tree can be pruned, starting at the root of the branches that represent the language constructions to be removed.

To identify those specific points, we found convenient to use the same metalanguage in which the to-be-reduced metamodel is expressed, thus following the example of aspect weaving where the syntax of the “aspectized” language is reused for expressing advices. Pruned points in an Ecore (or MOF) to-be-reduced metamodel are identified in an Ecore (or MOF) reduction metamodel: the elements to be cut are duplicated in the unmerge metamodel using the same name and included in a matching hierarchy. Thus, prune points are identified as leaves of the reduction metamodel. Since the pruning points are matched with elements of the to-be-reduced metamodel according to their name, and since the metamodeling language is directly used to define a change in a metamodel, the mechanism looks like package merge. As we aim to reduce a metamodel rather than extending it, we decided to name this approach package unmerge.

In order to unmerge metamodels in a deterministic way, we had to define a composition hierarchy of concepts, and matching rules. This hierarchy is defined as follows:

- the root is a package,
- a package may contain other packages and classes,
- a class may contain properties or invariant constraints,
- properties and invariant constraints do not contain other elements.

An element in the reduced metamodel will match an element in the unmerge metamodel if they both have:

- the same name,
- the same metaclass (i.e. packages can only match packages, classes can match only classes, etc.),
- matching owners.

Constraints can be either strengthened or relaxed. If a leaf element has a stronger constraint, then the matching element appears in the final metamodel (i.e. is not re-

moved) but updated with this stronger constraint. Elements that hold constraints are the following:

- classes, that can be either concrete or abstract; an abstract class being more constrained than a concrete class,
- properties, that may define multiplicities; a property with a smaller multiplicity range is considered more constrained than a property with a larger multiplicity range.

Invariant constraints deserve special attention as they may be either explicitly or implicitly dropped. Indeed, since some elements are removed in the metamodel resulting from the unmerge, remaining invariant constraints (i.e. invariant constraints that did no match invariant constraints explicitly specified in the unmerge metamodel) are all typed-checked, and dropped if checking does not succeed.

Finally, the name of the metamodel resulting from the unmerge transformation is the name of the unmerge metamodel.

Package unmerge proceeds the following way. All the elements of the to-be-reduced metamodel that match leaf elements of the package unmerge metamodel are recursively removed from the original metamodel. Removing a class C also removes the properties whose type is C. Moreover, if a C class inherits from a B class to be removed, and if B inherits from classes A1 and A2 to be kept, then C class in the reduced metamodel will inherit from classes A1 and A2 directly. Leaf elements from the unmerge metamodel that do not match any element in the metamodel to be unmerged are ignored. As such, if unmerging a metamodel L with an unmerge metamodel U will produce a metamodel L--, unmerging L-- again with the U metamodel will result in the L-- metamodel: the unmerge transformation is idempotent.

The algorithm for unmerging a metamodel is defined as follows. For sake of space and readability, opposite properties and re-affectation of properties type is not discussed in this paper.

*Algorithm to find a matching element*

```
match(MM, e): elt
  elt ← ∅
  MM.elements.each{ ue |
    e.name = ue.name
    && e.metaType = ue.metaType
    && match(MM, e.owner) = ue.owner
  } ⇒ elt ← ue}
```

*Algorithm to unmerge a metamodel MM with an unmerge metamodel MM<sub>u</sub>*

```
packageUnmerge(MM, MMu) : MMt, MMm
  1. Copies source meta-model MM into target metamodel MMt and its merge MMm
  MMt ← MM, MMm ← MM, Ereq ← {}, Emerge ← {}
  2. Checking types
  MMt.types.each{ t |
    2.1 Types are kept in MMm and removed from MMt except if...
    match(MMu, t) ≠ ∅ ⇒ Emerge ← Emerge ∪ {match(MMm, t)}
```

```

2.1.a it is abstract in  $MM_u$  while not in  $MM$ 
if !t.isAbstract && match( $MM_u$ , t).isAbstract then
     $E_{req} \leftarrow E_{req} \cup \{t\}$ , t.abstract = true
2.1.b it removes not all properties / not all constraints
elsif match( $MM_u$ , t).eStructuralFeatures  $\neq \emptyset$ 
    || match( $MM_u$ , t).constraints  $\neq \emptyset$  then
         $E_{req} \leftarrow E_{req} \cup \{t\}$ 
2.1.c If a class is removed, its sub-classes
else
    - are kept in merge
     $E_{merge} \leftarrow E_{merge} \cup t.subClasses.each\{ mme \mid$ 
        match( $MM_m$ , mme)  $\}$ 
    - inherit from its super-class
    t.subClasses.each{ s | s.superClasses  $\leftarrow$ 
        (s.superClasses / {t})  $\cup t.superClasses$  }
end if
2.2 Types that are not in unmerge are kept only in target meta-model
match( $MM_u$ , t) =  $\emptyset \Rightarrow E_{req} \leftarrow E_{req} \cup \{t\}$  }

```

### 3. Checking properties and constraints

```

( $MM_t.types.structuralFeatures \cup MM_t.types.constraints$ ).each{ p |
    3.1 Properties and constraints from  $MM_u$  are kept in  $MM_m$  and removed from  $MM_p$ ,
    except if...
    match( $MM_u$ , p)  $\neq \emptyset$ 
         $\Rightarrow E_{merge} \leftarrow E_{merge} \cup \{match(MM_m, p), match(MM_m, p.owner)\}$ 
    3.1.a the element is a property with a different multiplicity
    p  $\in p.owner.structuralFeatures$ 
        && ( p.lower  $\neq$  match( $MM_u$ , p).lower
            || p.upper  $\neq$  match( $MM_u$ , p).upper))
         $\Rightarrow (E_{req} \leftarrow E_{req} \cup \{p\}, E_{req} \leftarrow E_{req} \cup \{p.owner\},$ 
            p.lower  $\leftarrow$  max(p.lower, match( $MM_u$ , p).lower),
            p.upper  $\leftarrow$  min(p.upper, match( $MM_u$ , p).upper))
    3.2 Types that are not in unmerge are kept only in target meta-model
    match( $MM_u$ , p) =  $\emptyset \Rightarrow E_{req} \leftarrow E_{req} \cup \{p\}$  }

```

### 4. sub-packages

```

 $MM_t.subPackages.each\{ sp \mid$ 
    4.1 Packages in unmerge are kept in merge and removed from target metamodel,
    except if...
    match( $MM_u$ , sp)  $\neq \emptyset \Rightarrow E_{merge} \leftarrow E_{merge} \cup \{match(MM_m, sp)\}$ 
    4.1.a it removes not all contents
    match( $MM_u$ , sp).types  $\neq \emptyset$ 
         $\Rightarrow ((sp_u, sp_m) \leftarrow packageUnmerge(sp, match(MM_u, sp)),$ 
             $E_{req} \leftarrow E_{req} \cup \{sp_u\}, E_{merge} \leftarrow E_{merge} \cup \{sp_m\},$ 
             $MM_t \leftarrow (MM_t / \{sp\}) \cup sp_u, MM_m \leftarrow MM_m \cup \{sp_m\})$ 
    4.2 Packages that are not in unmerge are kept only in target meta-model
    match( $MM_u$ , sp) =  $\emptyset \Rightarrow E_{req} \leftarrow E_{req} \cup \{sp\}$  }

```

### 5. Remove non-required elements in target meta-model

(elements include sub-packages, types, and constraints)

```

 $MM_t.elements.each\{ e \mid e \notin E_{req} \Rightarrow MM_t.elements \leftarrow MM_t.elements / \{e\}\}$ 

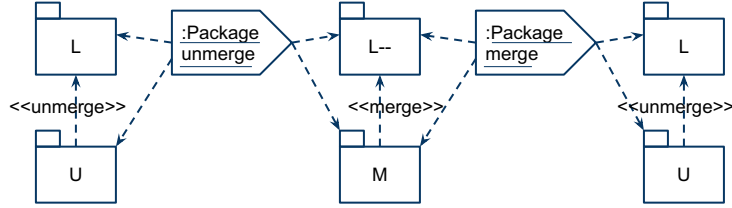
```

#### 6. Checking constraints

```
MMt.elements.each{ e | e.constraints.each{ c | !c.compile()
  ⇒ (e.constraints ← e.constraints / {c},
    Emerge ← Emerge U {match(MMm, e)} ) }
```

#### 7. Remove non-required elements in merge meta-model

```
MMm.elements.each{ e | e ∉ Emerge ⇒ MMm.elements ← MMm.elements / {e} }
```



**Fig. 1.** Reversibility of the package merge and package unmerge transformations.

As shown in Figure 1, the outcome of an unmerge is the reduced version of the original metamodel (L--). While the unmerging transformation removes some elements from a metamodel, the dual package merge transformation adds elements to a metamodel. Interestingly, package merge and unmerge transformations can also generate the counterparts which may be used later to undo the effect of either merge or unmerge. Hence, in addition to the resulting metamodel, the transformation may reference all those concepts that were removed from the metamodel to be unmerged (L) in an extension taking the shape of a package merge (M).

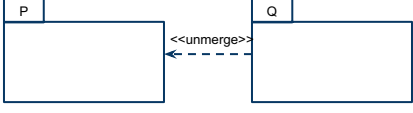
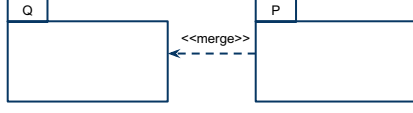
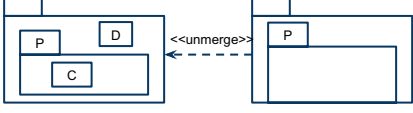
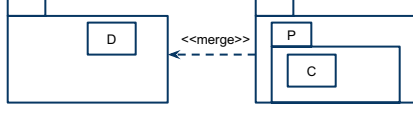
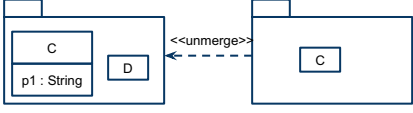
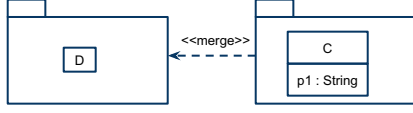
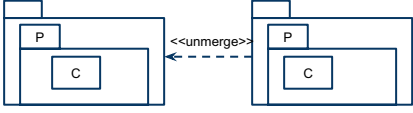
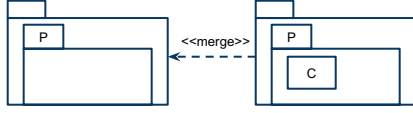
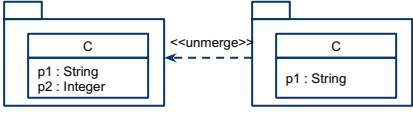
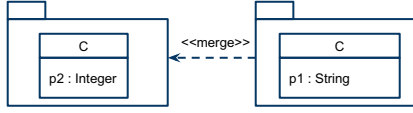
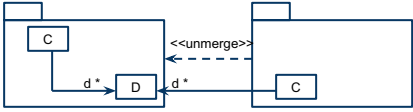
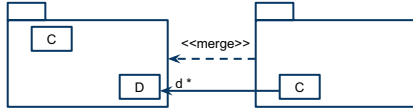
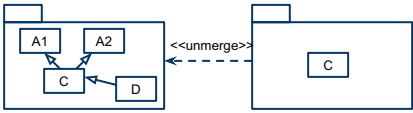
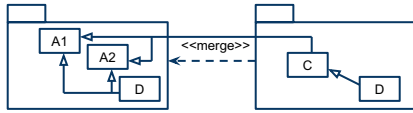
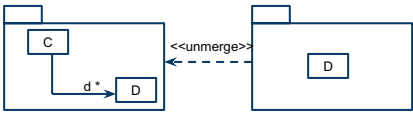
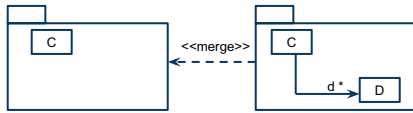
To go back to the unmerged metamodel (L), one just needs to perform a package merge transformation on the unmerged version (L--) driven by the previously generated merge (M). Thus, the generated merge (M) plays the role of the trace of the unmerge transformation: it makes it possible to control what happened during the unmerge, and to reverse the unmerge process.

Symmetrically, the package merge transformation can be extended to generate the unmerge counterpart, so that any addition to the merged metamodel (L--) is referenced in a generated unmerge counterpart (U). As such, the package unmerge transformation is the inverse transformation of the package merge transformation.

## 5 Unmerge use cases

This section presents in Table 1 a set of simple examples which illustrate the main aspects of the package unmerge relation. As explained in the previous section, since any reduction defined in package unmerge has its package merge counterpart, we also show the counterparts to be merged in order to go back to the original metamodel.

**Table 1.** Unmerge use cases

Unmerge use cases	Results and merge counterparts
<p>Unmerging package P</p> 	
<p>Removing package P</p> 	
<p>Removing class C</p> 	
<p>Removing class P : C</p> 	
<p>Removing attribute C . p1</p> 	
<p>Removing reference C . d</p> 	
<p>Removing class C in hierarchy</p> 	
<p>Removing referenced class D</p> 	



**Table 1.** Unmerge use cases

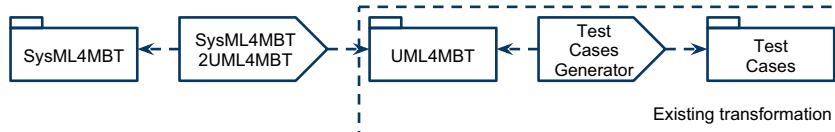
Unmerge use cases	Results and merge counterparts
<p>Making C class abstract</p>	
<p>Removing an invariant constraint of C</p>	
<p>Strengthening multiplicity for C . d</p>	

## 6 Example

This section shows how the package merge and unmerge relations may be used to build a metamodel by reusing other metamodels. The overall context is model-based testing [11] of SysML models, and the example is borrowed from the VETESS project. The goal is to generate a set of test cases from a behavioral model of the system under test.

The available tooling for test generation is based on a dialect of the UML language (called UML4MBT, UML for Model Based Testing), and a model transformation may be used to translate UML models to UML4MBT models (which is out of the scope of this example).

The same scheme is implemented for SysML models. A dedicated SysML dialect (called SysML4MBT, SysML for Model Based Testing) has been defined. A model transformation has been written to translate SysML4MBT models to UML4MBT models, thus allowing direct reuse of the tooling for test generation. Figure 2 describes this transformation chain.

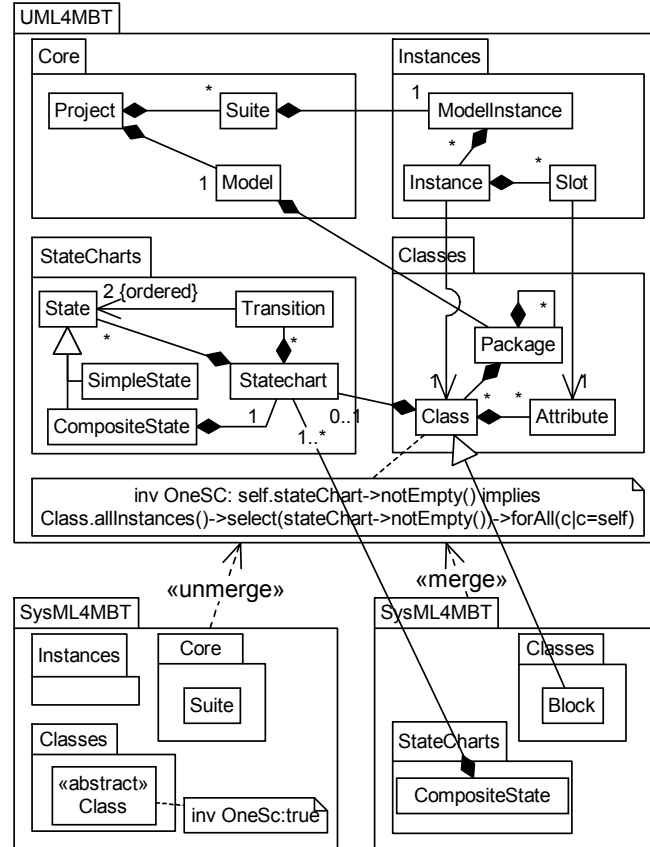


**Fig. 2.** VETESS tool chain for SysML.

SysML4MBT and UML4MBT are good examples of languages which are more or less similar. They share a lot of commonalities, but diverge on some parts. To specify a

model transformation between SysML4MBT and UML4MBT is it convenient to explicit how these two languages compare, and how they can be built from each other.

Figure 3 shows how SysML4MBT can be derived from UML4MBT. Constructions to be removed are represented in the package unmerge metamodel while parts to be added are specified in the package merge metamodel.



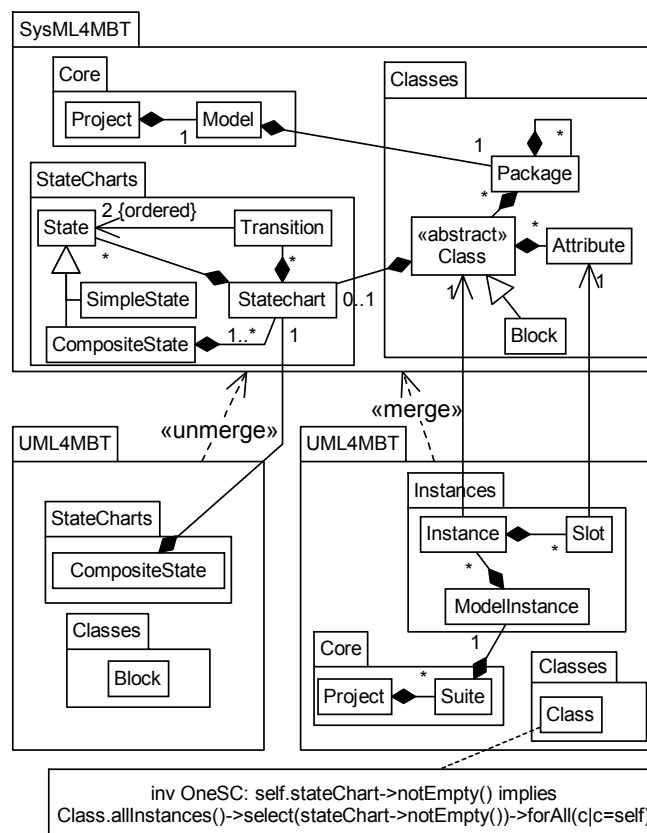
**Fig. 3.** Deriving SysML4MBT from UML4MBT.

The representation is interpreted as follows:

- The Instances package has to be removed, including all contained elements.
- The Core::Suite metaclass has to be removed. The containing Core package will not be removed, but references to Suite will be dropped (in our case Core::Project.suite).
- Class is made abstract (by the unmerge) and Block is added as a concrete sub-class (by the merge). Notice here that merging could not be used to set Class as abstract, because the merge semantics state that merging concrete with abstract yields concrete.

- The multiplicity of the composition relation between CompositeState and Statechart is set to 1..\* (by the merge).
- The constraint OneSC attached to Class is removed by the unmerge of a constraint of the same name (the body is not used for the unmerge; so we arbitrarily set it's value to true)

As explained earlier, package merge and package unmerge can be used either to trace their effect each other, or to undo their effect. We will illustrate this last point in the following lines. Figure 4 shows how to build UML4MBT from SysML4MBT, by merging and unmerging the respective counterparts.



**Fig. 4.** Deriving UML4MBT from SysML4MBT.

Again, the representation is interpreted as follows:

- The unmerge part states that `Classes::Block` should be removed, and that the `StateCharts::CompositeState::stateChart` multiplicity should be strengthened to 1..1.

- The merge part redefines `Instances` and its components, which are equivalent to those dropped from UML4MBT, re-introduces the `Core::Suite` construct (including incoming and outgoing references), makes the `Classes::Class` metaclass concrete, and re-introduces the `OneSC` constraint.

Package merge and package unmerge, along with the respective counterparts, can be used to go back and forth from one metamodel to another. From this point, it becomes possible to automate, at least partially, the translation from SysML4MBT to UML4MBT (and conversely from UML4MBT to SysML4MBT). If some parts of this transformation need a lot of engineering (e.g. transforming multiple parallel state machines into one single state machine), other parts are trivial. Indeed, because of the way SysML4MBT is produced from UML4MBT, those two metamodels expose many similarities. In the SysML4MBT to UML4MBT transformation, those similarities take the shape of “copy” rules: `SysML4MBT::Project` elements create `UML4MBT::Project` elements, `SysML4MBT::Model` create `UML4MBT::Model`, etc. Finally, any information whose structure in SysML4MBT was kept from UML4MBT is merely copied to the resulting UML4MBT model.

As such an automatic tool could be created to filter SysML4MBT models and translate them into UML4MBT models for those modeling elements whose structure is common between the two metamodels. The modeling elements in the SysML4MBT models that could not be transformed in such a way would be clearly identified, and would have to be treated independently by a “hand made” transformation. Such mechanism may be seen as a higher-order version of the  $\gg$  operator found in [12].

## 7 Related works

As mentioned in section 2, extending metamodels was paid much less attention than reducing. However, one interesting proposal was made by Sen et al. [1]. They identify four reasons to motivate the reduction of a metamodel and thus avoid over-specification:

- clearly state what are the input/output domains of a model transformation,
- avoid chaining transformations with inconsistent input/output domains,
- avoid generating input data models with unused concepts when testing transformations,
- avoid confusing a model designer.

They also propose an algorithm for reducing a metamodel. This algorithm requires the set of all interesting elements in a metamodel; those elements are kept in the resulting metamodel, including their dependencies in a transitive way. However, they do not state how interesting constructs can be identified. Our approach rather identifies elements that must not appear in the reduced metamodel. Indeed, identifying all interesting parts may require an effort as important as defining a metamodel from scratch. Moreover, we state how those “uninteresting elements” can be identified using the metalanguage in which the metamodel-to-be-reduced is defined. Finally, thanks to the symmetry that exists between the merge and unmerge relations, we are able to create the reverse definition to highlight what the reduction actually did.

Some aspect-oriented modeling techniques, such as MATA [10], provide means for deleting modeling constructs in class diagram-like models. A strength of these techniques is that they can designate various elements in a metamodel using a single rule. Such multiple designation rules can easily be integrated in package unmerge (and package merge), e.g. by matching regular expressions instead of merely comparing construct names. In our approach, we propose to clearly separate the notion of adding information from removing information in two distinct metamodels (for merge and unmerge). Moreover, transformation outcomes can be enforced right away in counterparts that use the same formalism (merge and unmerge metamodels), rather than inspecting weave traces afterwards. Another difference stems from the fact that our primary goal is to overcome the problems encountered when merging metamodels: if the reused metamodel is not as fine-grained as necessary, package unmerge comes into play to remove well identified elements.

Klein et al. [13] propose to unweave an aspect-oriented transformation of a model: added information may be later removed. Their approach relies on the analysis of the trace produced by a previously performed weaving. Interestingly, our approach does not require a previously executed action: the unmerge mechanism does not depend on a merge transformation, supposed to be performed before.

Metamodel matching [14] is another field related to our work. Metamodel matching compares two given metamodels and outputs a mapping that can be used to specify or generate a model alignment transformation [15]. Package merge and unmerge could be used to represent this mapping with emphasize on commonalties and differences.

Steel et al. [12] define rules for comparing two metamodels. This way model transformations may declare their input and output domains, so as to check that a given model can actually “enter” a transformation. As such, they check that a model which conforms to a given metamodel also conforms to another metamodel. Unfortunately, a model conforming to a reduced metamodel may not always conform to the metamodel-to-be reduced. This stems from the properties of the merge transformation. As pointed in [16], a model conforming to a metamodel-to-be-merged may not conform to the merged metamodel. As the counterpart of package merge, package unmerge may thus not preserve model typing. A concluding remark is that extending the perimeter of a language is not the only possibility of package merge; symmetrically, reducing the perimeter of a language may not be done only by package unmerge.

Package unmerge, following the example of package merge, is a metamodel manipulation technique: it implicitly defines a transformation for a metamodel. Instead of package merge or unmerge, one might rather consider using a general-purpose model transformation language, such as QVT [17] to remove information from a metamodel. However, our approach defines what one could call a domain-specific transformation language: package merge and unmerge only focus in transforming metamodels for a particular class of problem: reuse. Package merge and package unmerge contain much less constructs than general-purpose transformation languages, and are thus easier to learn and to use. Moreover, package merge and unmerge definitions use the same surface syntax as the transformed metamodels. It was shown in [18] how this last point can ease the creation of transformations: rules are much smaller, more readable, and no

knowledge about the meta-metamodel (i.e. the internal structure of the transformed metamodel) is required. Nevertheless, package merge and package unmerge may conveniently be implemented by model transformations written in general purpose transformation languages.

## 8 Conclusion

This work is a contribution to the field of metamodel reuse, in the context of language engineering. We have presented here a new mechanism for controlling metamodel reduction, based on the definition of counterparts to package merge relations, that we call package unmerge.

Package merge and package unmerge can be considered a dual approach to metamodel engineering, by which the effect of one can be traced and reversed by the other. Used together, package merge and unmerge allow fine tuning of metamodel reuse.

We have defined the operational semantics of the package merge transformation, and the corresponding algorithm is presented in the paper.

We have developed a tool which implements both package merge and unmerge, and which provides assistance to determine the subset of a metamodel that a given tool effectively implements. The tool also automates the generation of package counterparts for package merge and unmerge. This tool is open-source, and can be downloaded from <http://sourcesup.cru.fr/projects/vetess/>.

Future directions include the addition of pattern matching to package unmerge, so that reductions could be applied to several points in a metamodel, in one step. This would come very close to aspect-weaving (or better-said, aspect-unweaving [13]), and we would like to further investigate this point.

## References

- [1] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, “Meta-model pruning,” in Schürr and Selic [19], pp. 32–46.
- [2] E. Rahm and P. A. Bernstein, “A survey of approaches to automatic schema matching,” *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001.
- [3] P. Shvaiko and J. Euzenat, “A survey of schema-based matching approaches,” *Journal on Data Semantics IV*, vol. 3730, pp. 146–171, 2005.
- [4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, “A subset of precise UML for model-based testing,” in *A-MOST’07, 3rd int. Workshop on Advances in Model Based Testing*, (London, UK), pp. 95–104, ACM Press, July 2007.
- [5] Object Management Group, “Unified Modeling Language (UML), superstructure, version 2.1.2.” OMG Document formal/2007-11-02, November 2007.
- [6] Object Management Group, “Meta-Object Facility (MOF) core, v2.0.” OMG Document formal/2006-01-01, January 2006.

- [7] Object Management Group, “Unified Modeling Language (UML), infrastructure, version 2.1.2.” OMG Document formal/2007-02-06, November 2007.
- [8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” *ECOOP’97 - Object-Oriented Programming*, vol. 1241, pp. 220–242, 1997.
- [9] A. Schauerhuber, W. Schwinger, W. Retschitzegger, M. Wimmer, and G. Kappel, “A survey on aspect-oriented modeling approaches,” tech. rep., Vienna University of Technology, October 2007.
- [10] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, and J. Araújo, “MATA: A unified approach for composing UML aspect models based on graph transformation,” *T. Aspect-Oriented Software Development VI*, vol. 6, pp. 191–237, 2009.
- [11] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.
- [12] J. Steel and J.-M. Jézéquel, “On Model Typing,” *Journal of Software and Systems Modeling (SoSyM)*, vol. 6, pp. 401–414, 2007.
- [13] J. Klein, J. Kienzle, B. Morin, and J.-M. Jézéquel, “Aspect model unweaving,” in Schürr and Selic [19], pp. 514–530.
- [14] D. Lopes, S. Hammoudi, J. de Souza, and A. Bontempo, “Metamodel matching: Experiments and comparison,” in *ICSEA*, p. 2, IEEE Computer Society, 2006.
- [15] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, “Metamodel matching for automatic model transformation generation,” in *MoDELS* (K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, eds.), vol. 5301 of *Lecture Notes in Computer Science*, pp. 326–340, Springer, 2008.
- [16] J. Dingel, Z. Diskin, and A. Zito, “Understanding and improving uml package merge,” *Journal of Software and Systems Modeling (SoSyM)*, vol. 7, pp. 443–467, October 2008.
- [17] Object Management Group, “Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.” OMG Document formal/2008-04-03, April 2008.
- [18] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, “Towards model transformation generation by-example,” in *HICSS*, (Los Alamitos, CA, USA), p. 285b, IEEE Computer Society, 2007.
- [19] A. Schürr and B. Selic, eds., *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, vol. 5795 of *Lecture Notes in Computer Science*, Springer, 2009.