Big Metamodels are Evil

Package Unmerge - A technique for downsizing metamodels

Frédéric Fondement, Pierre-Alain Muller,
Laurent Thiry, Brice Wittmann,and Germain Forestier

MIPS, Université de Haute Alsace,
12, rue des frères Lumière - 68093 Mulhouse cedex - France
{frederic.fondement,pierre-alain.muller,
laurent.thiry,brice.wittmann,germain.forestier}@uha.fr

Abstract. While reuse is typically considered a good practice, it may also lead to keeping irrelevant concerns in derived elements. For instance, new metamodels are usually built upon existing metamodels using additive techniques such as profiling and package merge. With such additive techniques, new metamodels tend to become bigger and bigger, which leads to harmful overheads of complexity for both tool builders and users. In this paper, we introduce «package unmerge» - a proposal for a subtractive relation between packages - which complements existing metamodel-extension techniques.

1 Introduction and motivation

In the domain of software engineering, reuse is typically achieved by sharing reusable software parts in so-called libraries. From reusable procedures or structures, those parts evolved into fully fledged components [1]. Components are pieces of software that can be combined together to build up new software systems. Research related to this topic showed that it is of paramount importance to define precisely contracts for components, upon which both component makers and component users can rely [2]. Interface specification, which indicates what messages can be treated or sent by a component, is only the very first step towards the definition of a contract [3]. Of course, any component claiming to implement a contract must fulfill it completely.

Programming languages are another mean for helping software reuse. Indeed, languages abstract away details of platforms while still making it possible to describe expected behavior of a software system. Examples of platforms' details abstracted by many languages are the instruction set of a processor, and available interruptions of an operating system. A compiler can automatically infer details abstracted away from code so that an executable program can be delivered, as long as the code conforms to the expected programming language. This way, the same source code could be used by different compilers made for the same programming language, but targeting different platforms, e.g. different processors or operating systems. Model driven engineering (MDE) pushes the same idea a step further: abstracting away details of platforms while offering simple constructs in a modeling language, with compilers being replaced by model transformations. In addition to model transformations, a given model can be manipulated by a constellation of tools, thus following a data-centric architectural style.

Examples of such tools for modeling are textual or graphical model editors, model verifiers, model checkers, model serializers, model interpreters, and model transformation engines. All of them need to be able to manipulate one (or more) model(s). Following the example of compilers handling programs written in a well-defined language, tools for modeling handle models with a well defined structure. The model structure is most of the time specified by a metamodel. In this realm, metamodels are to modeling tools, what contracts were to components.

One typical problem with this data-centric architectural style, is that tools might not all have the same capabilities. If some tools may handle all of the possible modeling constructs of a language (as defined in a metamodel), some other tools may only work on a given subset of those constructs [4]. An example is UML code generators, which are usually able to generate code for class diagrams, or state chart diagrams, yet discarding any information provided by use case diagrams or timing diagrams. It does not necessarily mean that such tools, which cover only part of a modeling language, should be blamed: usually, discarded information is just useless within the perspective of the intent of the tools [5]. However, it might make tools' users feel that the contract of the tool is not fulfilled as parts of the metamodel are ignored.

A similar situation happens when it comes to defining a new language by reusing an existing metamodel. To reuse an existing metamodel, one usually extends it by defining additional concepts and relations. To extend a metamodel, one possible solution is to use an annotation mechanism such as profiles [6 - section 18]. However, this approach is an additive-only technique, making the resulting metamodel bigger than the extended metamodel. Thanks to (or because of) this additive nature, tools for the extended language can still work on models of the new language, yet ignoring all information that could be included in the model thanks to the extension. As an example, if someone extends UML with a profile, UML code generators will be still able to generate code for profiled UML models, but information carried by the profile application will be merely ignored, usually without even a warning.

From the modeling tools' users point of view, the only way to know whether an element of a model will be ignored or not is to read documentation written in natural language, or to try and see either nothing happening, or an error message raised while invoking the tools. This situation can be compared to a compiler not considering the complete program code without clearly stating which part of the code is considered. By indicating formally the subset of the metamodel that is actually covered, a tool could be made more precise regarding handled models, i.e. regarding its contract. One could thus check his/her models in order to know exactly what information is to be ignored by a given tool. Moreover, by supplying a real and clear interface (i.e. metamodel) of handled elements, modeling tools could be more easily selected, verified, or assembled, following the advanced practices of the software component community.

While metamodel extension has deserved significant interest, reduction has not yet gained the same exposure. As a consequence, the more a metamodel is built by reusing other metamodels, the more it is likely to contain irrelevant constructs from the perspective of a given tool. In other words, metamodels contain too many features, one reason for that being that it is currently impracticable to identify and remove unneeded parts.

In this paper, we examine how reduction of metamodels could be expressed in an explicit way, basically by describing package unmerge mechanism, built as a counterpart of the existing package merge metamodel additive extension mechanism as defined in [7 - section 11.9.3].

The paper is organized as follows: after this introduction, section 2 presents three dominant techniques for metamodel extension, section 3 presents our proposal for reduction (which we call package unmerge relation), section 4 describes the package unmerge algorithm, section 5 provides more in depth examples, section 6 gives a short overview of the tool support we propose for package merge and unmerge, section 7 compares our approach to others, and section 8 concludes and presents future directions.

2 Extending metamodels

Typical mechanisms for controlling metamodel extension include UML profiles, package merge relations, and aspect weaving.

Profiles [6 - 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 [8]) 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 conform to their original metamodels (regardless of profiles).

Package merge relations [7 - section 11.9.3], as opposed to profiles, are considered an heavyweight 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 [9]. 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 [10]. Many different formalisms have been studied so far, including UML class diagrams [11]. 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 metamodeling 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.

3 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, such structure is defined using a set of meta-classes and relations between meta-classes; a model can thus be seen as a set of related instances. By altering those meta-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.

To identify those specific removal points, i.e. elements that should be dropped from a to-be-reduced metamodel, we found convenient to use the same metalanguage in which the to-be-reduced metamodel is expressed. Meta-elements to be pruned in a to-be-reduced metamodel are identified in a 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. Corresponding elements in the to-be-reduced metamodel can thus be identified as to be removed. In addition, all elements part of the hierarchy of removed element should also be removed, even if not explicitly designated by 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 at reducing 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. Hierarchy and matching rules depend on the metalanguage used to define metamodels. This hierarchy is defined as follows:

- the root is a package,
- a package may contain other packages and classes,
- a class may contain properties and 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 removed) 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 also have to be updated according to pruning action performed on the metamodel-to-be-reduced. In the case of package unmerge, any invariant constraint depending on a pruned element (e.g. a metaclass or a property) should also be marked to be unmerged, otherwise the package unmerge definition would be illegal. In this, we follow our guideline to define package merge as the pure counterpart of package unmerge that can add new invariant constraints (but not relax existing ones).

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 properties whose type is C (see UC3 in Table 1 below). 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 (see UC7 in Table 1 below). Leaf elements from the unmerge metamodel that do not match any element in the metamodel to be unmerged are ignored.

Table 1 shows a set of simple example use cases which illustrate the main aspects of package unmerge. First column shows the to-be-reduced metamodel together with an unmerge metamodel, and second column shows the reduced metamodel obtained after unmerging, together with the merge metamodel necessary to get the original to-be-reduced metamodel back. This latter part is more extensively explained in the next section.

4 Unmerge Algorithm

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

Table 1. Unmerge use cases

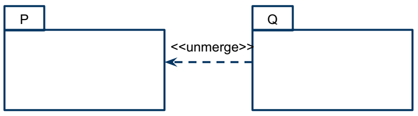
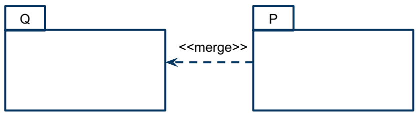
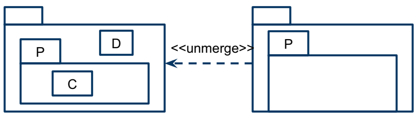
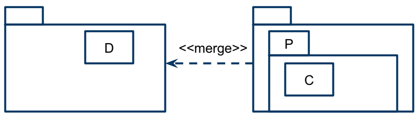
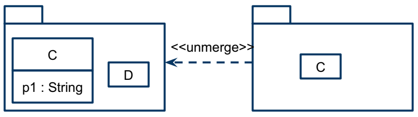
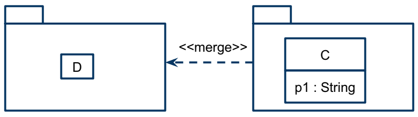
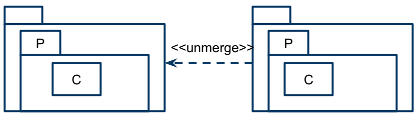
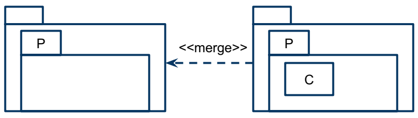
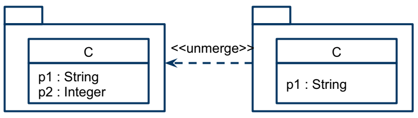
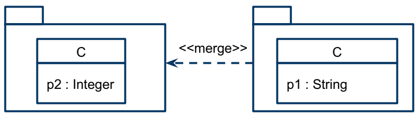
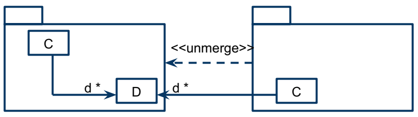
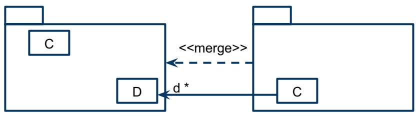
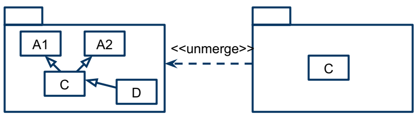
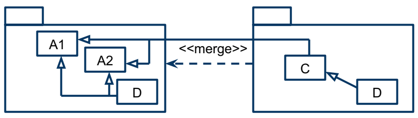
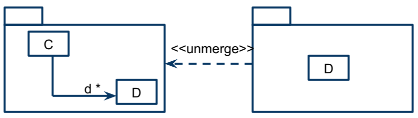
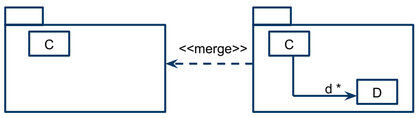
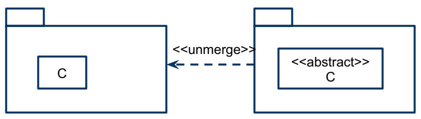
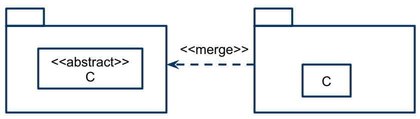
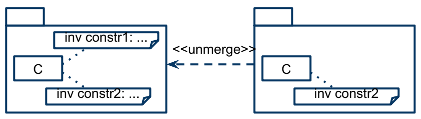
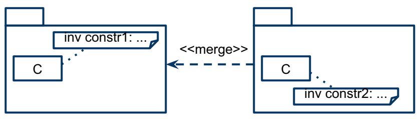
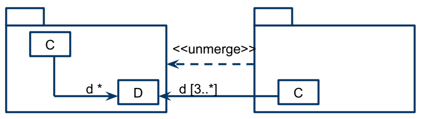
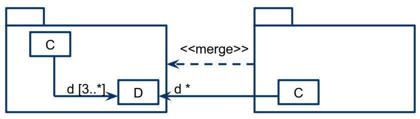
Unmerge use cases	**Results and merge counterparts**
UC1 - Unmerging package `P` 	
UC2 - Removing package `P` 	
UC3 - Removing class `C` 	
UC4 - Removing class `P::C` 	
UC5 - Removing attribute `C.p1` 	
UC6 - Removing reference `C.d` 	
UC7 - Removing class `C` in hierarchy 	
UC8 - Removing referenced class `D` 	

Table 1. Unmerge use cases

Unmerge use cases	Results and merge counterparts
UC9 - Making `C` class abstract 	
UC10 - Removing constraint of `C`  inv constr1: ... C inv constr2: ... <<unmerge>> C inv constr2	
UC11 - Strengthening `C.d` multiplicity 	

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).

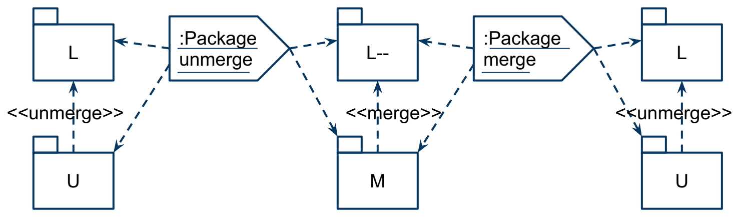


Fig. 1. Reversibility of the package merge and package unmerge transformations.

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. It also allows to reflect any eventual change in L-- or M back to L or U. 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.

The algorithm for unmerging a metamodel is defined here with the same formalism as in [4]. The algorithm relies on structure shown in Figure 2 for metamodels (MM, MM_u, MM_t, and MM_m), though it could be adapted to other class-oriented structures. `elements` is an operation returning recursively all composed elements from `Package`, and `removeElement` an operation that removes an element from a package wherever it occurs in the hierarchy of the package. For sake of space and readability, opposite properties and re-affectation of properties type is not discussed in this paper.

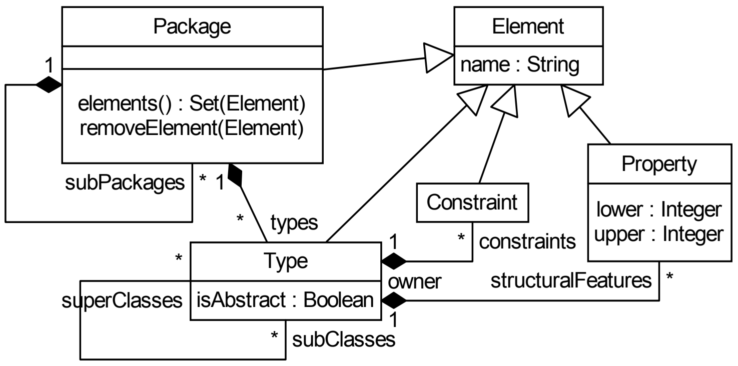


Fig. 2. Expectations on an unmerged metamodel

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_u

```
packageUnmerge(MM,$MM_u$) : $MM_t$,$MM_m$
     1. Copies source meta-model MM into target metamodel $MM_t$ and its merge $MM_m$
     $MM_t$ ← MM, $MM_m$ ← MM, $E_{req}$ ← {}, $E_{merge}$ ← {}
     2. Checking types
     $MM_t$.types.each{ t |
          2.1 Types are kept in $MM_m$ and removed from $MM_t$ (as C in UC3), except if...
          match($MM_u$, t) ≠ ∅ ⇒ $E_{merge}$ ← $E_{merge}$ ∪ {match($MM_m$, t)}
          2.1.a it is abstract in $MM_u$ while not in MM (as C in UC9)
          if !t.isAbstract && match($MM_u$,t).isAbstract then
               $E_{req}$ ← $E_{req}$ ∪ {t}, t.abstract = true
          2.1.b it doe not remove all properties/constraints (as C in UC5/UC10)
          elsif match($MM_u$,t).structuralFeatures ≠ ∅
               || match($MM_u$,t).constraints ≠ ∅ then
               $E_{req}$ ← $E_{req}$ ∪ {t}
          2.1.c If a class is removed (as C in UC7), its sub-classes (as D in UC7)
          else
```

```
            - are kept in merge (as D in $MM_m$ in UC7)
            $E_{merge} \leftarrow E_{merge} \cup$ t.subClasses.each{ mme |
                  match($MM_m$,mme)}
            - inherit from its super-class (D inherits A1 and A2  in UC7 $MM_t$)
            t.subClasses.each{ s | s.superClasses ←
                     (s.superClasses / {t}) ∪ t.superClasses}
      end if
      2.2 Types not in unmerge are kept only in target meta-model (as D in UC3)
      match($MM_u$, t) = ∅ ⇒ $E_{req} \leftarrow E_{req} \cup$ {t} }

3. Checking properties and constraints
$MM_t$.types.structuralFeatures ∪ $MM_t$.types.constraints).each{ p |
      3.1 Properties and constraints from $MM_u$ are kept in $MM_m$
      and removed from $MM_t$ (as C.p1 in UC5), except if...
      match($MM_u$, p) ≠ ∅
            ⇒ $E_{merge} \leftarrow E_{merge} \cup$ {match($MM_m$,p), match($MM_m$, p.owner)}
      3.1.a the element is a property with a different multiplicity (as C.d in UC11)
      p ∈ p.owner.structuralFeatures
            && ( p.lower ≠ match(MMu, p).lower
                  || p.upper ≠ match($MM_u$, p).upper))
            ⇒ ($E_{req} \leftarrow E_{req} \cup$ {p}, $E_{req} \leftarrow E_{req} \cup$ {p.owner},
                  p.lower ← max(p.lower, match($MM_u$, p).lower),
                  p.upper ← min(p.upper, match($MM_u$, p).upper))
      3.2 Elements that are not in unmerge are kept only in target meta-model
      match($MM_u$, p)= ∅ ⇒ $E_{req} \leftarrow E_{req} \cup$ {p} }

4. sub-packages
$MM_t$.subPackages.each{ sp |
      4.1 Packages in unmerge are kept in merge
      and removed from target metamodel (as P in UC2), except if...
      match($MM_u$, sp) ≠ ∅ ⇒ $E_{merge} \leftarrow E_{merge} \cup$ {match($MM_m$, sp)}
      4.1.a it removes not all contents (as P in UC4)
      match($MM_u$, sp).types ≠ ∅
            ⇒ (($sp_u$, $sp_m$) ← packageUnmerge(sp, match($MM_u$, sp)),
                  $E_{req} \leftarrow E_{req} \cup$ {$sp_u$},   $E_{merge} \leftarrow E_{merge} \cup$ {$sp_m$},
                  $MM_t$ ← ($MM_t$ / {sp}) ∪ $sp_u$, $MM_m \leftarrow MM_m \cup$ {$sp_m$})
      4.2 Packages that are not in unmerge are kept only in target meta-model
      match($MM_u$, sp) = ∅ ⇒ $E_{req} \leftarrow E_{req} \cup$ {sp} }

5.Remove non-required elements in target meta-model
(elements include sub-packages, types, and constraints)
$MM_t$.elements().each{ e | e ∉ $E_{req}$ ⇒ $MM_t$.removeElement(e)}
6.Remove non-required elements in merge meta-model
$MM_m$.elements().each{ e | e ∉ $E_{merge}$ ⇒ $MM_m$.removeElement(e)}
```

5 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 of SysML models, and the example is borrowed from the VETESS project [12]. 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 as explained in [13]. Figure 3 describes this transformation chain.

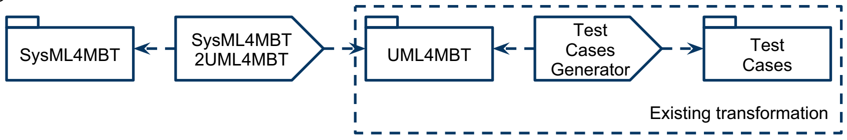


Fig. 3. VETESS tool chain for SysML.

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, it is convenient to explicitly state how these two languages compare, and how they can be built from each other.

Figure 4 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.

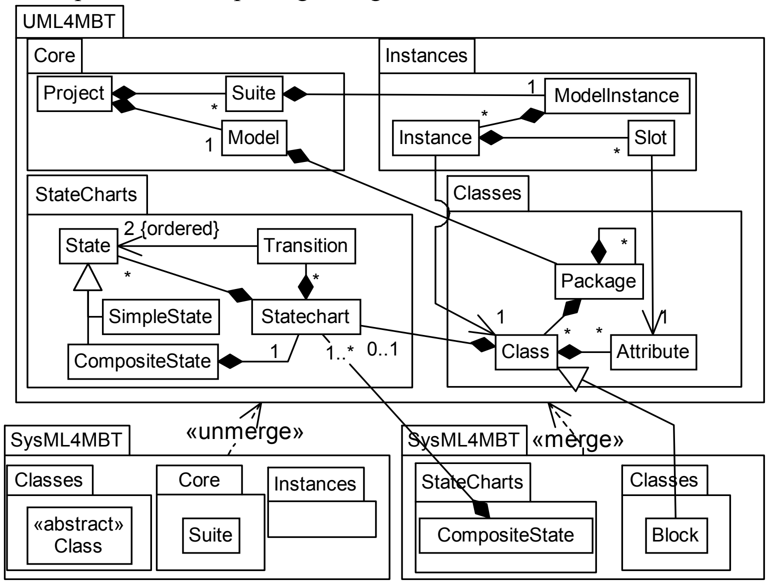


Fig. 4. Deriving SysML4MBT from UML4MBT.

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 subclass (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).

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 5 shows how to build UML4MBT from SysML4MBT, by merging and unmerging the respective counterparts.

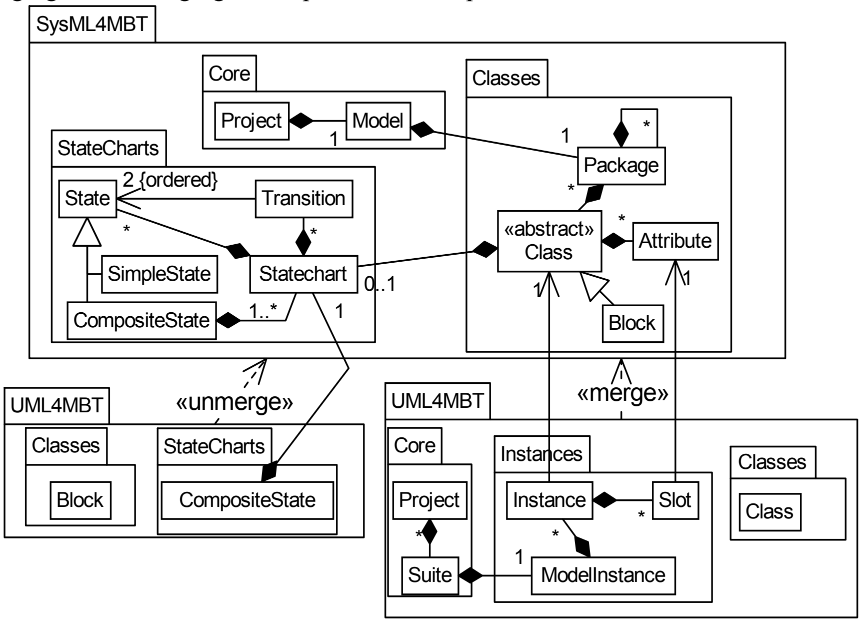


Fig. 5. Deriving UML4MBT from SysML4MBT.

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), and makes the `Classes::Class` metaclass concrete.

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). 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.

6 Implementation

An open-source prototype implementation for both package merge and package unmerge as it is defined in this paper is available on the project website[1]. This implementation takes the shape of an Eclipse plug-in. A set of tutorials, corresponding to Table 1., is also available from the VETESS website[2].

The transformation can be invoked on an ecore file holding serialization for a package unmerge metamodel. The metamodel to be unmerged is referenced in an annotation of the package unmerge metamodel. Once invoked, the file name for serializing the unmerged metamodel is given and the transformation happens. The outcome is another ecore file for the unmerged metamodel plus an additional ecore file representing the merge counterpart. We developed a similar transformation for package merge, which also produces the merged metamodel and the corresponding unmerge counterpart.

While experimenting UML metamodel unmerge, we found it a very repetitive and error-prone task to eliminate invariant constraints (i.e. UML well formedness rules) that depend on an element that was pruned in the unmerged metamodel. That is why we included in the prototype a drop mechanism that discards from the unmerged metamodel any invariant constraint on which a type checking could not succeed.

In order to exchange models, we also developed automatic model transformation so that a model conforming to an unmerged metamodel can be transformed into a model conforming the metamodel that was unmerged. We also made the reverse transformation that removes information from a model of a metamodel that was unmerged so that it becomes a model of the unmerged metamodel. These two transformation make it possible to reuse existing models and interact with a tool defined for working on a metamodel subset defined in terms of package merge.

1. *https://sourcesup.cru.fr/projects/vetess/*

2. *http://bit.ly/litERM*

7 Related works

As mentioned in section 1, reducing metamodels was paid much less attention than extending. However, one interesting proposal was made by Sen et al. [4]. 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. To sum up, our approach fits better when a lot of top elements are to be removed, and when complex operations are necessary (such as removing a class from an inheritance hierarchy, or making a class abstract). Otherwise, approach of [4] fits better when only a few elements are to be kept in a metamodel, all of them being well identified.

Some few aspect-oriented modeling techniques, whose purpose is to weave changes into a (meta)model, provide means for deleting modeling constructs as a "removal" advice. One example is MATA [11] for 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 could easily be integrated in package unmerge (and package merge), e.g. by introducing more sophisticated pattern matching constructs. Compared to aspect-oriented modeling, package merge and unmerge clearly separate the notions of adding information from removing information in two distinct specifications. Another difference is that package unmerge is one simple additional relationship construct to be added to metamodeling languages, unlike aspect-oriented modeling which requires completely new languages, even if aspect languages are defined as extensions to the languages to which aspects are to be applied. Such extensions include additional concepts to the base language (like pointcuts, a set of designators, and different categories of advice). Those extensions could be described by means of package merge and unmerge.

Metamodel matching and differencing [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]. Differences can be shown in a difference model (such as an AMW model [16]) that would represent the equivalent for our package unmerge model. First, package merge and unmerge could be used as alternative models to represent this mapping while emphasizing commonalties and differences. Second, the difference model usually references the compared (meta)models. As such, it is not possible to compute one metamodel from the other as both need to exist. However, instead of merely relying on a named elements hierarchy, package merge and package unmerge could benefit from metamodel matching techniques to match elements of the package (un)merge with elements of the package-to-be-(un)merged.

(Meta)model slicing [17] is a technique taking its roots in program slicing and graph decomposition. It makes it possible to extract from a model (and thus a metamodel) a sub-model containing elements depending on a set of elements of interest. The set of elements to be kept is computed from transitive dependencies of the elements of interest, and finally, only those elements that are not related to the elements of interest are discarded. Package unmerge rather identifies elements to be removed, and all contained elements are also removed, even if a dependency exists between an element to be kept and an element to be removed. An example found in section 5 was the `Core::Suite` that had to be dropped even though `Core::Project` had to be kept. Purpose of model slicing is more about model understanding and impact analysis while purpose of package unmerge is metamodel reuse.

Steel et al. [18] 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 out in [19], 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.

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 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/*.

Package unmerge, due to its definition as the package merge counterpart, inherits its strengths and drawbacks from package merge. It designates clearly what is to be removed, which may be an advantage (no unexpected removal) or a drawback (different pruning points all have to be designated). Moreover, as package merge is not the only mechanism for composing metamodels, the package unmerge we propose here is not be the only approach to metamodel pruning. For example, in [20], beside metamodel merge (corresponding to the approach taken by package merge approach) are identified metamodel interfacing, class refinement, and template instantiation. Counterparts for some of these approaches might also be possible and deserve to be explored and compared to, now package unmerge is proposed a definition.

We consider a metamodel too big when it is used by a tool that does not handle all of the concepts it declares. Making clear what actual metamodel is used by modeling tools would make tools' behavior clearer, as metamodel of manipulated models is part of tools' contract. A problem with many tools is that they do not fulfill their contract, because they declare a metamodel that is often too big, especially for metamodels constructed by reuse. One solution for this problem is to be able to alter extended metamodels using subtractive techniques as the one we propose in this paper. Thus, we consider metamodel reduction as step towards what one could call «component-based model engineering», where modeling tools could be selected, verified or assembled according to their contract. Hopefully, shifting to component-based paradigm could change the nature of MDE as components changed the nature of software [21].

References

[1] C. A. Szyperski, *Component software - beyond object-oriented programming*. Addison-Wesley-Longman, 1998.

[2] B. Meyer, *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988.

[3] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Making components contract aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[4] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, "Meta-model pruning," in *MoDELS* (A. Schürr and B. Selic, eds.), vol. 5795 of *Lecture Notes in Computer Science*, pp. 32–46, Springer, 2009.

[5] P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale, "Modeling modeling modeling," *Software and System Modeling*, vol. 11, no. 3, pp. 347–359, 2012.

[6] Object Management Group, "Unified Modeling Language (UML), superstructure, version 2.4.1." OMG Document formal/2011-08-06, August 2011.

[7] Object Management Group, "Unified Modeling Language (UML), infrastructure, version 2.4.1." OMG Document formal/2011-08-05, August 2011.

[8] Object Management Group, "Meta-Object Facility (MOF) core, v2.4.1." OMG Document formal/2011-08-07, August 2011.

[9] 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.

[10] 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.

[11] 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.

[12] J. Lasalle, F. Peureux, and F. Fondement, "Development of an automated MBT toolchain from UML/SysML models," *ISSE*, vol. 7, no. 4, pp. 247–256, 2011.

[13] J. Lasalle, F. Bouquet, B. Legeard, and F. Peureux, "SysML to UML model transformation for test generation purpose," in *UML&FM'10, 3rd IEEE Int. Workshop on UML and Formal Methods*, (Shanghai, China), pp. 1–8, 2011.

[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] M. Didonet, D. Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the Eclipse AMW plugin," in *In Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[17] H. H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in *ICSM*, pp. 635–638, IEEE Computer Society, 2005.

[18] J. Steel and J.-M. Jézéquel, "On Model Typing," *Journal of Software and Systems Modeling (SoSyM)*, vol. 6, pp. 401–414, 2007.

[19] 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.

[20] M. Emerson and J. Sztipanovits, "Techniques for metamodel composition," in *in The 6th OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2006*, pp. 123–139, ACM, ACM Press, 2006.

[21] P. Herzum and O. Sims, *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise*. Wiley, 1999.