Package Merge in UML 2: Practice vs. Theory? *

Alanna Zito, Zinovy Diskin and Juergen Dingel

School of Computing, Queen's University
Kingston, Ontario, Canada
{zito,zdiskin,dingel}@cs.queensu.ca

Abstract. The notion of compliance is meant to facilitate tool interoperability. UML 2 offers 4 compliance levels. Level L_{i+1} is obtained from Level L_i through an operation called *package merge*. Package merge is intended to allow modeling concepts defined at one level to be extended with new features. To ensure interoperability, package merge has to ensure *compatibility*: the XMI representation of the result of the merge has to be compatible with that of the original package. UML 2 lacks a precise and comprehensive definition of package merge. This paper reports on our work to understand and formalize package merge. Its main result is that package merge as defined in UML 2.1 does not ensure compatibility. To expose the problem and possible remedies more clearly, we present this result in terms of a very general classification of model extension mechanisms.

1 Introduction

Since UML is intended to support systems engineering in general, its scope is extremely broad. Potential application domains include not only software and hardware engineering, but also data and business process engineering. Particular domains may only require certain features of UML, while others may be completely irrelevant. To support its use in different domains, UML was designed in a modular fashion: Modeling features are defined in separate, and, as much as possible, independent units, called *packages*.

To support exchange of models and interoperability between UML tools, UML 2 partitions the set of all its modeling features into 4 horizontal layers called *compliance levels*. Level L_0 only contains the features necessary for modeling the kind of class-based structures typically encountered in object-oriented languages. Level L_3, on the other hand, encompasses all of UML. It extends level L_2 with features that allow the modeling of information flows, templates, and model packaging. According to the UML 2.1 specification, a tool *compliant at level* L_i must have "the ability to output diagrams and to read in diagrams based on the XMI schema corresponding to that compliance level" [11, Section 2.3]. Moreover, to achieve interoperability, the tool must be *compatible* with tools at

* Research supported by IBM CAS Ottawa and OCE Centre of Communications and Information Technology

lower compliance levels; that is, it must be able to load all models from tools that are compliant at lower levels, without loss of information.

The precise definition of the compliance levels in UML 2.1 rests on a novel operation called *package merge*. Informally, package merge is intended to allow concepts defined in one package to be extended with features defined in another. The package defining level L_{i+1} is obtained from L_i by merging new features into the package describing L_i. For instance, the level L_1 package is created by merging 11 packages (e.g., *Classes::Kernel*, *Actions::BasicActions*, *Interactions::BasicInteractions*, and *UseCases*) into the level L_0 package. In the UML 2.1 specification, package merge is described by a set of transformations and constraints grouped by metamodel types [11, Section 7.3.40]. The constraints define pre-conditions for the merge, such as when two package elements "match", while the post-conditions are given by the transformations. The merge of two packages proceeds by merging their contents as follows: Two matching elements are merged recursively. Elements that do not have a matching counterpart in the other package are deep-copied. The specification describes the general principle behind package merge as follows [11, Section 7.3.40, page 116]:

> "a resulting element will not be any less capable than it was prior to the merge. This means, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge."

In the same paragraph, the specification states that package merge must ensure compatibility in the sense that the XMI representation of the resulting package must be compatible with that of the original package. This "compatibility property" of package merge is crucial. It guarantees that a tool T compliant at some level is compatible with all tools compliant at lower levels, because it allows T to load models created with lower level tools without loss of information.

Theoretically at least, package merge may be useful not only for the definition of the UML metamodel, but also for users of UML in general. However, a more thorough evaluation of package merge, not to mention a more general adoption, is not straight-forward:

- The detailed semantics of package merge is currently only discussed for certain types found mostly in metamodels (e.g., classes, associations, and properties). It is not clear how to extend package merge to other types such as interactions and state machines.
- The semantics of package merge is perceived as complicated. For instance, the UML manual describes it as "complex and tricky" and recommends the use of package merge only for "metamodel builders forced to reuse the same model for several different, divergent purposes" [12, p. 508]. One reason for this perception may be that the general intent of package merge is not clear. The general principle in the specification cited above is too imprecise.

The long-term goal of our work is to study the general principles underlying package merge. The goal of this paper is to report on the first results of our work. In particular, we will present a general classification of package extension mechnisms based on a convenient notational and terminological framework. Moreover, the application of this classification to package merge in UML 2 will allow us to conclude that:

1. Package merge as *defined* in the UML 2.1 specification does not ensure compatibility; that is, the XMI representation of the result of the merge is not necessarily compatible with the XMI representation of the original package.
2. It appears that package merge as *used* for the definition of the compliance levels of UML 2.1 does ensure the compatibility property.

After providing the necessary background and briefly reviewing related work in the next section, Section 3 will present the general classification. Section 4 will describe its application to package merge. Section 5 will conclude and outline further work.

2 Background

2.1 Package merge

To illustrate package merge, consider a simple class model of employees as shown in package *BasicEmployees* on the left of Figure 1.

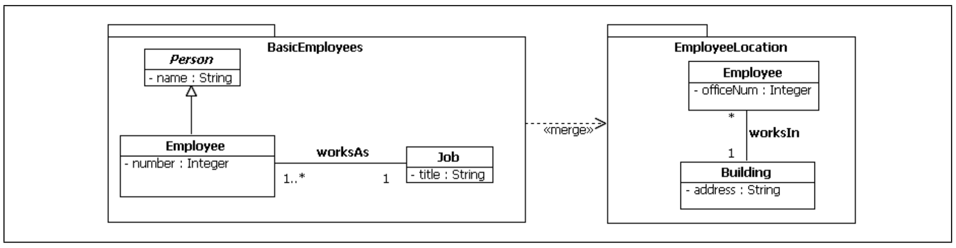


Fig. 1. A package merge example

Suppose we want to extend this model with the information an office manager might have as shown in package *EmployeeLocation*. To this end, package *EmployeeLocation* is merged into package *BasicEmployees*, as indicated by the arrow in between the two packages in Figure 1. We say that *BasicEmployees* is the *receiving package*. Its elements (classes *Person*, *Employee*, and *Job* and the association *worksAs*) are called *receiving elements*. *EmployeeLocation* is the *merged package*. Its elements (classes *Employee* and *Building* and association *worksIn*) are called the *merged elements*. The *resulting package* is shown in Figure 2. It is obtained by merging the merged elements into the receiving package. Since the class *Employee* in *EmployeeLocation* matches the class of the same name in *BasicEmployee*, the two are merged recursively by adding the property *officeNum* to the receiving class. The class *Building* and the association

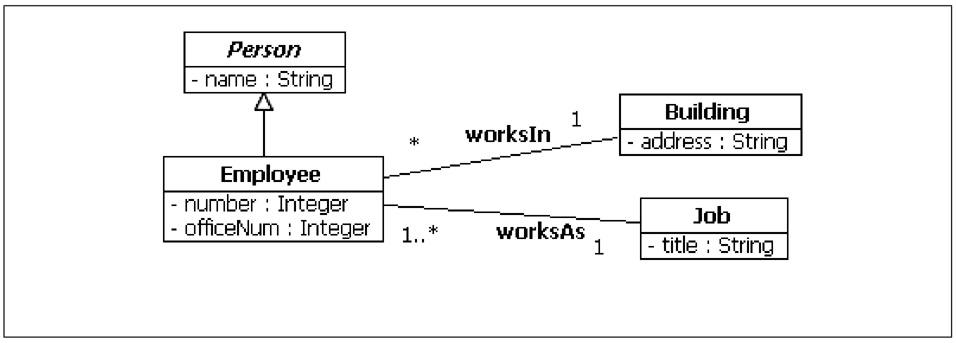


Fig. 2. The result of the merge in Figure 1

worksIn, however, do not match any elements in the receiving package and are simply copied. Note that the ≪*merge*≫ arrow merely implies these transformations and that the resulting package is actually not shown in Figure 1.

2.2 Compatibility

The UML specification clearly states that all model extension mechanisms used to define compliance levels such as package merge must have the "compatibility property" mentioned above. This property asserts the compatibility of the XMI representation of the resulting package with that of the receiving package and thus ensures that a tool is compatible with all tools compliant at lower levels. We refine the definition of the "compatibility property" as follows. We say that *package A is compatible with package B*, if every document allowed by the XML Schema of B is also allowed by the XML Schema of A. We refer to the constraints expressable in XML Schema (e.g., the elements and attributes that can appear in a document, and the order and number of child elements) as *compatibility constraints*. Additional constraints that a package may contain such as OCL constraints are referred to as *validity constraints*.

2.3 Related work

According to [13], package merge was partially inspired by two specification combination mechanisms offered in Catalysis: "and" and "join". However, while related on first glance, both differ substantially from package merge. The "and" operation is for use with subtyping, while the "join" operation allows a specification to "impose additional preconditions to those defined in another view" [9, p. 697].

Speaking in more precise terms, two issues are to be distinguished in package merge. The first is the merge procedure as such. In this context, package merge is a particular case of a known problem in databases and, more recently, the Semantic Web. About twenty years ago this problem was referred to as *view/schema* *integration* [3]; its more recent name is *model merge* [5]. Package merge is a typical example of the schema integration problem when schema matching is easy (because schemas to be merged were designed by the same team) and based on name coincidence.

The second issue is an evaluation of the merge result: whether it is good or bad w.r.t. the goals of package merge. The primary criterion here is level compliance; that is, in more detail, compatibility of the legal instances of the receiving package with the resulting package (as a metadata schema). It follows then that we need to evaluate the relationship between the resulting P' and receiving P packages in terms of sets of their instances. This issue is well-studied in mathematical logic and model theory under the name of *theory extension* (and yes, *theory* is one more synonym for our term *package*; definition and basic results can be found in any textbook on mathematical logic, see e.g., [2]). This observation is essential for package merge, since it is a well-known fact that extensions can be *non-conservative*; that is, new data/structure added to the theory can influence the "old" part of the theory in such a way that not all old instances can be augmented with new structure (be compatible with the new structure). It shows that package merge mechanism as such does not guarantee, in general, compatibility between the packages and a more thorough investigation is needed.

The notion of theory as it is formulated in mathematical logic is heavily based on a specific syntax (logical connectives and quantifiers) and is not suited for package merge studies. The same dependance on syntax also prevents the use of many results obtained in schema/model integration for package merge. We need a more abstract framework, and here the so-called *institutions*, introduced in [10] and now well known in the algebraic specifications community, provide the necessary instrumentary. Our theoretical considerations in sect.3 are inspired by institutions and, in fact, just adjust the definitions to the PM context (see also [6, 1] for a similar elaboration in other contexts).

The issues of package merge and extension can be seen in a even wider context as particular problems in *generic model management*, a prominent program that has recently appeared in databases [4] and is rapidly broadening its agenda towards a general theory of model manipulation and transformation (see [8] for a survey).

3 Theoretical foundations via examples.

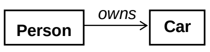
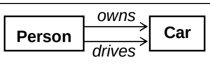
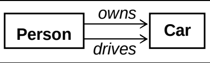
In this section we describe a general framework for package merge and extension. Here we use the term package as a generic term meaning either a data model/schema, or XML Schema, or metadata model, or, in general, any object P having an associated set of instances, $inst(P)$.

Our plan is as follows. We will begin with considering a series of generic examples of package merge (PM), presented in Tables 1, 2, to outline the scope of the issue. While discussing these examples, we offer a convenient terminology and notation to encode them. Particularly, we show that relations between the (sets of) instances of the receiving and the resulting packages constitute the essence of the compatibility issues in package merge. Moreover, we will develop a taxonomy of these relationships and demonstrate how it works. Then we elaborate the notation in more precise terms and, in fact, make it ready for formalization. The latter is omitted due to space limitations.

3.1 Basic terminology, definitions and taxonomy of package extensions

In formal terms, *package merge* (PM) is an operation that takes two packages, P_1, called the *merged* package, and $P2$, the *receiving* package, then integrates their contents in a certain way, and assigns the name P_2 to the result. In the usual programming language notation, it can be written as $P_2 := P_1 + P_2$. The intuition of incremental increase of the content of the receiving package suggests another notation, $P' = P + \varDelta$, which we will follow further.

Table 1. Generic examples of package extensions

1	2	3				4	5
Package	Pure structural base of the package	Multiplicities for associations				Correspondences between associations	Compatibility of old instances and, if not, their fixability
		owns		*drives*			
		left	right	left	right		
P		1..*	1..3	N/A		N/A	N/A
Δ(L,R)		**2..4**	**2..4**	L	R	Context *P:Person*: *P.drives* ⊆ *P.owns*	N/A
P'(L,R)= P+Δ(L,R)		**1..***	**1..4**	same		same	Depends on (L,R)
	A few versions of merge with different values of parameters (L,R)						
P'₁	same	same		**1..2**	**0..3**	same	***Optional*** extension: **all** old instances are compatible
P'₂₁	same	same		same	**1..***	same	***Mandatory*** extension: **none** of old instances is compatible. Yet there is an option to fix ... (i) all of them
P'₂₂	same	same		same	**2..***	same	(ii) some of them
P'₂₃	same	same		same	**5..***	same	(iii) none of them (as extension is inconsistent)
P'₃	same	*****	same	*****	same	same	(iv) none of them but the extension is consistent
P'	Structure/Schema, ***S***	Compatibility constraints, C_C				Validity constraint, C_V	Type of extension: Conservative, non-conservative, inconsistent, totally non-conservative

Consider Table 1. The top row presents a (piece of some) receiving package P, and the second row is the merged package Δ; the resulting package is shown in the third row. The contents of each package participating in the table are separated into three parts: the structural base (graph of classes and associations), multiplicities (left and right) for the participating associations and correspondences (constraints) between associations (columns 2..4). The merged package Δ is parameterized by the left, L, and right, R, multiplicities of the association *drives*. Since there is no association *drives* in the receiving package, this association together with its multiplicities L, R will be copied to the resulting package; hence, the latter is also parameterized by L, R. As for multiplicities for association *owns*, according to the PM-rules, the resulting multiplicity has the lowest lower bound and greatest upper bound of the receiving and merged multiplicities.. This explains the $P'(L, R)$ row in the table.

The next five rows present five samples of merge differing only in the values of parameters, mainly in the right multiplicity for *drives*. These quantitative changes, however, cause all five cases to be qualitatively different w.r.t. relations between the sets of package instances, $\mathit{inst}(P)$ and $\mathit{inst}(P')$ respectively. We will call the elements of the former *old* instances and those of the latter *new*. In the 1st example (package P'_1), since the right end of association *drives* has the optional multiplicity, all old instances can be well considered as new instances and we have inclusion $\mathit{inst}(P) \subset \mathit{inst}(P')$. In such cases, we will say that all old instances are *compatible* with the new package structure, and that package P' is an *optional extension* of package P.

The next four rows present cases when none of the old instances can be loaded into the new package structure; in other words, sets $\mathit{inst}(P)$ and $\mathit{inst}(P')$ are disjoint. We will say that package P' is a *mandatory* extension of P.

If an instance I of package P is incompatible, we may try to fix it by adding missing items, in our case, missing *drives*-links. As examples $P'_{21}, P'_{22}, P'_{23}$ show, we can encounter situations when (i) *all*, (ii) *some* or (iii) *none* of the old instances are fixable in this sense. Note that package P'_{23} is totally *inconsistent* (has no instances), which of course implies that none of the old instances can be loaded into it. The case (iv) in the P'_3-row is more interesting. There, the left multiplicity of the *owns*-association for the merged package $\Delta(L, R)$ is changed from 2..4 to *. This multiplicity will go to the result (by the same PM- rule described above), and then package P'_3 – in contrast to package P'_{23} – is consistent: it does have instances consisting of *Car*-objects only. However, none of the old instances (of package P) can be fixed to become one of these P_3-instances. Correspondingly, we call the three subtypes (i,ii,iv) of *consistent* mandatory extensions *conservative, non-conservative and totally non-conservative* while in case (iii) the extension is itself *inconsistent.*

Among these three, the case of non-conservative extension (P'_{22}) is the most interesting. In general, the new constraints in package P' are statements about new items in the structure. Often, they relate these new items with the old ones (those in package P) like, for example, the correspondence statement in Table 1. The question is whether such statements can somehow constrain the old structure embedded in the new structure. In other words, let us take an instance $I' \in \mathit{inst}(P')$ of the resulting package, and forget about its additional structure, thus coming to a instance $I = \overleftarrow{I'} \in \mathit{inst}(P)$ of the receiving package. Let $\mathit{inst}^{\leftarrow}_{\Delta}(P)$ (where Δ refers to the package extension in question) denote the set of all such reduct-instances. At first glance, it may seem that the equality $\mathit{inst}^{\leftarrow}_{\Delta}(P) = \mathit{inst}(P)$ should hold but, in general, this is not the case. The point is that the new structure together with new constraints may be such a strong imposition over its old structure subset that not every old instance can be a reduct of some new instance. This phenomenon is well studied in mathematical logic and model theory under the name of non-conservative extension of theories (see, e.g., [2]). In the package merge context (where packages are, in fact, theories in a special graph-based logic [7]), it means that (for a mandatory package extension), not every old instance can be fixed to become compatible with the new structure (and hence be loaded into it).

Table 2 presents a simple example of non-conservative package extension (in the bottom row); some details for the extension P'_{22} in Table 1. It clearly shows that while every new instance can be mapped to an old instance by forgetting about its new extra structure (move from the right to the left in every row of the table), the inverse mapping is only partially defined (and, of course, is multivalued). In more formal terms, for any mandatory package extension $P' = P + \Delta$, a *forgetful* or *reduct* mapping $\mathsf{red}_{\Delta} : \mathit{inst}(P') \to \mathit{inst}(P)$ (to be read "forget Δ") is always defined but is not surjective. The inverse multivalued mapping $\mathsf{ext}_{\Delta} : \mathit{inst}(P) \to \mathit{inst}(P')$ (read "fix it by extending with Δ") is only partially defined. Note that in general we need to consider two versions of extending mappings: one is the (incomplete) extension of old instances towards compatibility into new package, the other is their complete extension to *valid* instances of P'. The example in the $I_2 - J_2$ row of Table 2 shows that these two mappings can be fundamentally different.

3.2 Unification and notation

All the examples above can be considered in some unified way and conveniently specified as follows. Let us consider a package as a triple $P = (S, C_C, C_V)$ where:

– S is some *structure* or *schema* having a certain set of instances, $\mathit{inst}(S)$. A typical example of a schema is a graph underlying a UML class diagram (nodes are class names and edges are associations), whose instances are specified by object diagrams over this schema. Another example is the tree structure of an XML document declared in its DTD, where its instances are all possible XML documents with this structure.

– C_C is a set of constraints regulating *compatibility* (hence the subscript ^{C}) of instances with the structure (think of UML multiplicities for associations or DTD constraints specifying optionality of elements in the XML document).

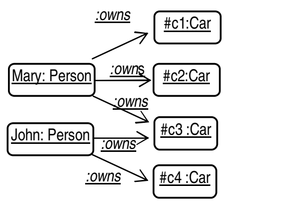
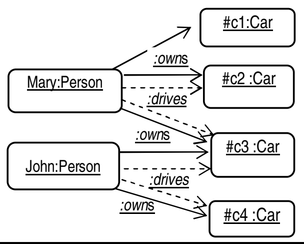
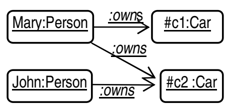
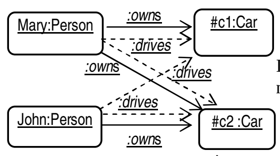
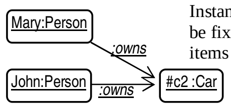
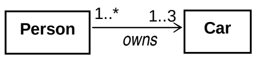
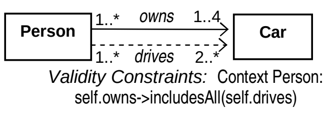
– C_V is an additional set of constraints specifying (in, say, OCL) *valid* instances among the compatible ones.

Since constraints narrow the set of instances, a package has three sets of instances associated with it:

$$inst_V(P) \subset inst_C(P) \subset inst_B(P) \overset{\text{def}}{=} inst(S)$$

which we will call, respectively, *valid, compatible and basic* instances of P. For example, instance J_2 in Table 2 is a compatible but not valid instance of package P', while instance J_3 is valid (and hence compatible).

Table 2. Example of non-conservative package extension (some details for the row P'_{22} in Table 1). The table is to be read from bottom to top.

Instances of package *P*	Instances of package *P'*$_{22}$	
	compatible	valid
...	...	...
 :owns #c1:Car Mary: Person :owns #c2:Car :owns John: Person #c3 :Car :owns :owns #c4 :Car instance I_3	 #c1:Car :owns Mary:Person #c2 :Car :drives :owns #c3 :Car John:Person :drives :owns #c4 :Car instance J_3	
 Mary:Person :owns #c1:Car :owns John:Person :owns #c2 :Car instance I_2	 Mary:Person :owns #c1:Car :drives :owns :drives :drives John:Person #c2 :Car :owns instance J_2	Instance J_2 is compatible but not valid.
 instance I_1 Instance I_1 is not compatible and cannot be fixed, i.e., augmented with missing items to become compatible.	No instances J_1 (that would become I_1 after forgetting their *drives*-links)	
 Package *P*	 Person 1..* owns 1..4 Car 1..* drives 2..* Validity Constraints: Context Person: self.owns->includesAll(self.drives) Package *P*$_{22}$' is a *mandatory* and *non-conservative* extension of *P*	

Correspondingly, package extension is described by the expression $P' = P + \Delta$ with the increment $\Delta = (\Delta_S, \Delta_C, \Delta_V)$ consisting of three component increments: in pure structure, Δ_S, in compatibility constraints to it, Δ_C, and in validity constraints to compatible instances Δ_V (we write Δ_C,Δ_V).

There is a delicate and important issue in specifying increments for constraints. The example in the top three rows of Table 1 shows that the merged package Δ can (i) change the compatibility of items in the (old) structure in the receiving package (multiplicities for association *owns*) and, of course, (ii) specify compatibility of new items added to the structure in the resulting package (multiplicities for *drives*). Thus, in general, $\Delta_C := \Delta_C + \Delta_C^*$, and similarly for Δ_V, where the $*$-index near Δ refers to constraints talking about the new items in the new structure while Δ without this index refers to changes in constraints for old items in the new structure.

We will also assume that our Δ's are always positive increments (addition), and to specify a decrement we write $(-\Delta)$. In more formal terms, it means that the sets of structures and constraints are partially ordered and, for example, $S' = S + \Delta$ means that $S \subset S'$ while $S' = S - \Delta$ means that $S' \subset S$ in that partial order on structures (normally, an ordinary sub-structure relation). Similarly, $C' = C + \Delta$ means that we strengthened the set of constraints by either adding new constraints to it or, maybe, by strengthening some of the constraints in C. In this notation, for example, the relation between packages P and P'_{22} in Table 2 can be specified by the following equalities: $S' = S + \Delta_S$, $C'_C = C_C - \Delta_C + \Delta_C^*$, $C'_V = C_V + \Delta_V^*$.

4 Applications to UML 2.1 Compliance Levels

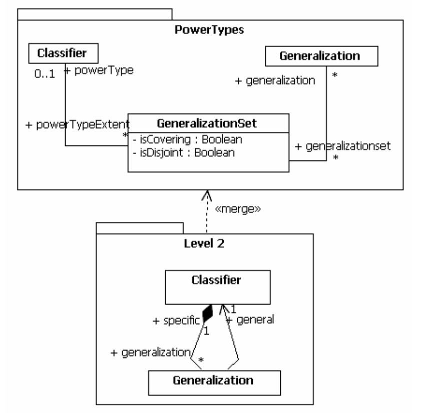
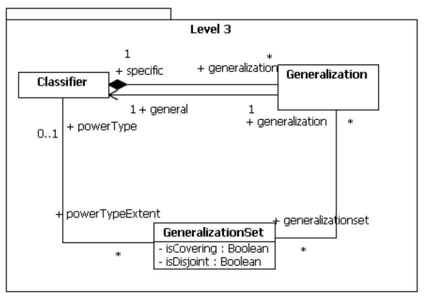
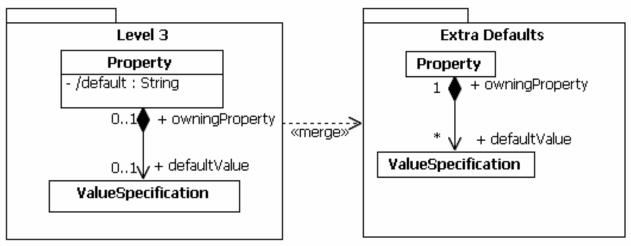
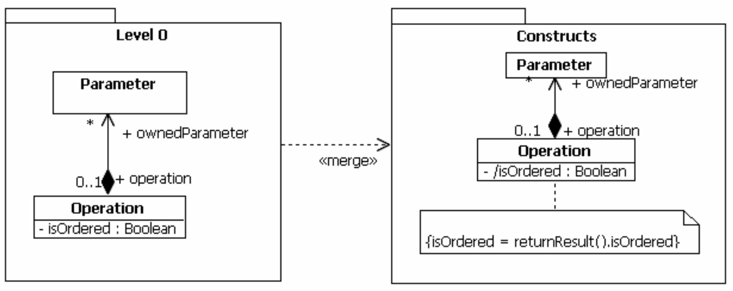
Having described and classified the general forms that package extension can take, we can now consider in more detail how the theory applies to package merge and the definition of UML compliance levels. The resulting package of a package merge can extend the receiving package in different ways, depending on the contents of the merged package. Some of these ensure compliance level compatibility, while others do not. We present here several examples of how package merge is used to define the compliance levels of UML 2.1, and show how each fits into the taxonomy of package extensions introduced in the Section 3. The taxonomy can be briefly summarized in pseudo-code as

```
if (extension is optional) then
  all instances are compatible
else //extension is mandatory
  if (extension is conservative) then
    all instances are not compatible, but can be fixed
  else if (extension is non-conservative) then
    if (extension is totally non-conservative) then
      no instances are compatible or fixable
    else
      no instances are compatible, but some are fixable
```

4.1 Examples of Optional Extension

The top row of Table 3 shows an example of a package merge taken from the definition of UML compliance levels in [11]. The *Level 2* package shows the relation-

Table 3. Examples of package merge resulting in optional extension

 PowerTypes Classifier 0..1 + powerType Generalization + generalization * + powerTypeExtent * GeneralizationSet - isCovering : Boolean - isDisjoint : Boolean + generalizationset * «merge» Level 2 Classifier + specific 1 1 + general + generalization * Generalization	 Level 3 1 + specific * + generalization Generalization Classifier 1 + general 1 + generalization * 0..1 + powerType + powerTypeExtent + generalizationset GeneralizationSet - isCovering : Boolean - isDisjoint : Boolean * *
1a. Optional extension	1b. Resulting package of (1a)
 Level 3 Property - /default : String 0..1 + owningProperty 0..1 + defaultValue ValueSpecification «merge» Extra Defaults Property 1 + owningProperty * + defaultValue ValueSpecification	
2. Optional extension with weakened compatibility	
 Level 0 Parameter * + ownedParameter 0..1 + operation Operation - isOrdered : Boolean «merge» Constructs Parameter * + ownedParameter 0..1 + operation Operation - /isOrdered : Boolean {isOrdered = returnResult().isOrdered}	
3. Optional extension with strengthened validity	

ship between the metaclasses *Classifier* and *Generalization* at level 2 compliance. One of the packages merged in to form level 3 compliance is the *PowerTypes* package, which contains additional structure for *Classifiers* and *Generalizations*. These additional elements (one class and two associations) are copied into the resulting package (which is also shown in Table 3), since they do not have matching elements in the receiving package. According to the definitions introduced in the previous section, the resulting package is an *optional* extension of the receiving package, meaning that it ensures that level 2 models are compatible with level 3-compliant tools.

The resulting package does not always add extra structure to the receiving package; it may only change the compatibility or validity constraints of the existing structure. This is a special case of optional extension. The second row of Table 3 shows two examples; since the resulting packages for both have the same structure as the receiving, they are not shown. The rules for package merge are such that existing compatibility constraints are always made less strict (weakened). For example, imagine that we want to add more features to UML with a fourth level of compliance. In Table 3, the *Level 3* package contains part of the definition of the metaclass *Property* at level 3 compliance. The (imaginary) *Extra Defaults* package defines that a *Property* may have more than one default value. According to package merge rules, the associations *owningProperty:Property* → *defaultValue:ValueSpecification* in the merged and receiving package will match, and their matching association ends will be recursively merged. The resulting *defaultValue* end has a multiplicity of 0..*, which is calculated by taking the lowest lower bound and the highest upper bound from the merged and receiving ends. A level 3-compliant model will thus never lose information when being imported into a level-4 compliant tool, since the resulting multiplicity is wider (weaker) than that of the receiving. It is interesting to note that, while the rules for merging multiplicities ensure the compatibility of the receiving and resulting packages, they also allow for previously illegal instances to be legal in the resulting package. For example, consider merging multiplicities 1..2 and 5..7. The resulting multiplicity will be 1..7, which includes values (i.e., 3..4) that were not allowed in either of the two original multiplicities. This is a consequence the fact that, as defined in UML 2.1, multiplicities must be a single, continuous interval.

Validity constraints, on the other hand, can be strengthened as the result of a merge. Section 3 introduced the notion of valid instances of a package as those instances which are compatible and which satisfy any additional semantic constraints on the model. The rules for package merge do not guarantee that instances of lower compliance levels will remain valid at higher levels. For example, Table 3 shows a package *Level 0*, which contains a part of the definition of operations at compliance level 0. One of the packages merged in to form level 1 compliance is the *Infrastructure::Constructs* package, which contains an identical definition of *Operation*, but with the additional constraint that the orderedness of an operation is derived from its return value. According to the rules for package merge, the constraint on the merged element *Operation* is added to the constraints on the receiving element Operation. The resulting package has the same structure as the receiving package, but its validity constraints have been added to (strengthened). If a model created at compliance level 0 does not derive the *isOrdered* property of its operations from their return parameters, it could be imported into a level 1-compliant tool, but would not fulfill all semantic constraints of that level.

4.2 Examples of Mandatory Extension

The resulting package of a package merge can also be a mandatory extension of the receiving. Figure 3 shows an example from [11]. The *Level 1* package contains a part of the definition of an *ActivityEdge* at that compliance level. The *IntermediateActivities* package, which is merged in to form level 2 compliance, contains an association *ActivityEdge* $\rightarrow$ *guard:ValueSpecification*, which has a non-optional (i.e., non-zero) multiplicity. A model created at level 1 compliance would not be compatible with a level 2-compliant tool, since its activity edges do not have a corresponding guard; it could not be imported "as-is" into the tool. However, since the extension in this case is conservative, it is possible to fix a non-compatible instance. The UML 2.1 specification explicitly states [11, Section 12.3.5], that the default value for the guard is "true", so a non-compatible model can be fixed by simply adding the default value as a guard on all *ActivityEdges* which do not have one.

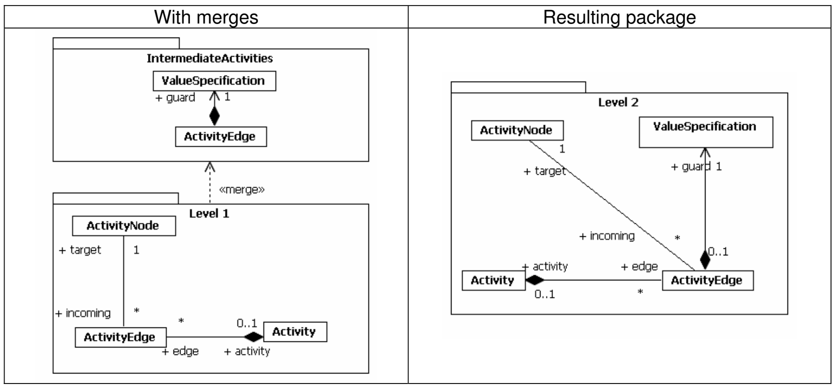


Fig. 3. Example of package merge resulting in mandatory extension

It is also possible for a package merge to result in a *non-conservative* mandatory extension of the receiving package. Although we do not present an example of this situation in terms of UML compliance levels, it is illustrated in examples P'_{21}, P'_{22} and P'_{23} of Table 1.

4.3 Summary

In terms of the notation introduced in Section 3, the rules for package merge ensure the following properties:

1. $S' = S + \Delta_S$, structure is never removed;
2. $C'_C = C_C - \Delta_C + \Delta^*_C$, compatibility constraints to the old structure are always weakened,
3. $C'_V = C_V \pm \Delta_V + \Delta^*_V$, validity constraints to the old structure can be strengthened or weakened.

Based on these rules, the relation of the receiving package to the resulting can range over our entire taxonomy of package extension. To the best of our knowledge, most of the uses of package merge for defining compliance levels of UML 2.1 result in optional extensions, and are thus compatible. The few instances of mandatory extension are conservative with fixes explicitly defined. However, in general, using package merge to define compliance levels for MOF-based models does not guarantee that successive levels will be compatible.

5 Conclusion and Future Work

UML 2.1 introduced the operation of package merge to faciliate the definition of compatible compliance levels. In order to better understand package merge, we have developed a theory of package extension, which is based on viewing an extension to a package as being made up of a pure structural increment, a compatibility constraint increment, and a validity constraint increment. This theory leads us to a taxonomy of the possible relationships between the original and extended package. The taxonomy distinguishes between optional extensions, which ensure compatibility, and mandatory extensions which do not. Mandatory extensions can be further subdivided into conservative, non-conservative and totally non-conservative extension, based on whether all, some or none of the original instances can be fixed to become compatible. This theory is influenced by concepts in model theory and mathematical logic, where theory extension has been well-studied. Our classification of package extension types allowed us to look at the relationship between the receiving and resulting package of a package merge in a more formal way. We have discovered that the rules for package merge do not prevent mandatory extension, and thus, it cannot guarantee compatibility when used to define compliance levels. However, in the definition of the compliance levels of UML 2.1, it appears that package merge is used in such a way as to ensure compatibility.

Future work on this topic includes completing a full formalization of our theory of package extension, as well as a formal definition of package merge. We are also interested in examining the notion of "fixability" in more depth to determine some sort of general guidelines or canonical way to fix incompatible models (where possible). Finally, another potential area of study is the impact of this work on UML modeling tools - for example, the ideal tool would have to

assess the compatibility and validity of models imported from tools of a lower compliance level, as well as suggest possible ways of fixing incompatible models.
Acknowledgements We would like to thank Bran Selic and Jim Amsden for taking the time to answer our questions about packge merge.

References

[1] S. Alagic and P. Berstein. A model theory for generic schema management. In *Eighth International Workshop on Databases and Programming Languages*, pages 228–246, 2001.

[2] J. Barwise, editor. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1977.

[3] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[4] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[5] P. Bernstein and R.Pottinger. Merging models based on given correspondences. In *Proc. Very large databases, VLDB'2003*, 2003.

[6] Z. Diskin. Abstract metamodeling, I: How to reason about meta-metamodeling in a formal way. In K. Baclawski, H. Kilov, A. Thalassinidis, and K. Tyson, editors, *8th OOPSLA Workshop on Behavioral Specifications, OOPSLA99*. Northeastern University, College of Computer Science, 1999.

[7] Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal arrow foundations for visual modeling. In *Diagrams'2000: 1st Int. Conf. on the Theory and Applications of Diagrams*, Springer LNAI#1889, pages 345–360, 2000.

[8] Zinovy Diskin and Boris Kadish. Generic model management. In Doorn, Rivero, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005.

[9] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML*. Addison Wesley, 1999.

[10] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.

[11] Object Management Group. *Unified Modeling Language: Superstructure (version 2.1, ptc/06-01-02)*, January 2006.

[12] I. Jacobson J. Rumbaugh and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 2 edition, 2004.

[13] B. Selic, January 2006. Personal communication.