Modeling UML 2 Package Merge with Alloy

Alanna Zito
IBM Rational Software
Kanata, Ontario, Canada
alanna_zito@ca.ibm.com

Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT

Package merge is a new modeling concept introduced into the latest version of the UML standard to help structure the UML metamodel. It is a relationship between two packages, where the contents of one package are merged into the contents of the other. Despite its importance in UML, package merge is not well understood.

As part of our work towards understanding and improving package merge in particular and model merge in general, we used Alloy to formalize and analyze different versions package merge. The analysis of the original version reveals the unexpected failure of crucial properties.

Categories and Subject Descriptors

D.2 Software Engineering [**D.2.1 Requirements/Specifications**]: Languages, methodologies, and tools; D.2.2 Design Tools and Techniques []

General Terms

Design, Verification

Keywords

Model merge

1. INTRODUCTION

The Unified Modeling Language (UML) is a general purpose modeling language that includes a wide variety of static and behavioural modeling concepts. A new version of the UML standard, UML 2, was recently adopted to incorporate advances in software modeling and modeling language design into the language [16, 14]. In addition to making obvious changes to user-level modeling concepts, such as adding new diagram types, the designers of UML 2 also made many changes to the UML *metamodel*. A metamodel describes (amongst possibly some other things) the syntax of a modeling language — what elements can appear in a model, and what kind of relationships can exist between them. The UML metamodel is described by a subset of UML itself, roughly corresponding to class diagrams; it consists of classes and associations grouped into packages. One of the new concepts added to UML 2 to help define the metamodel in a more structured and modular way is *package merge*.

Package merge is a directed relationship between two packages. It indicates that the contents of the target package are merged into the contents of the source package. The basic merge procedure is simple: any elements (for example, classes, associations, operations) in the target package that do not have a corresponding element with the same name in the source package are simply copied into the source package. If two elements have the same name, they are collapsed together into one element and their features are combined recursively. Package merge is particularly useful for extending a basic definition for different purposes; the base definition can be given in one package and extensions to this definition can be defined in other packages. The extensions can then be merged into the base package as appropriate to obtain a complete definition [16]. Package merge is used extensively in the UML metamodel. The metamodel is organized into three *compliance levels* where each successive level is defined by using package merge to extend its predecessor. In order to promote interoperability between different UML tools, these levels must be defined such that a tool on a higher level of compliance is *backwards compatible* with a tool on a lower level of compliance, that is, the higher level tool should be able to load models created by a lower level tool without loss of information.

Despite its importance to UML, package merge has not been studied in sufficient depth; the UML specification describes its semantics in informal terms and has been described as "complex and tricky" [17]; very few properties are discussed and the notion of backwards compatibility remains vague and unproven. While the goal of a more incremental and modular definition of UML is laudable, the current definition of package merge is not as useful as it could be, because "its use can lead to confusion and should be avoided if possible" [17].

To remedy this situation, we have formalized and analyzed the current version of package merge, and have identified problems, properties, and improvements. The fact that package merge does not ensure backwards compatibility has been reported in [24]. A list of ambiguous and missing merge rules is provided in [23]. In the present paper, we describe the formalization of the orginal version of package merge and its analysis with respect to certain standard, yet impor-

First Alloy Workshop. Portland, OR, USA. November 6, 2006
.

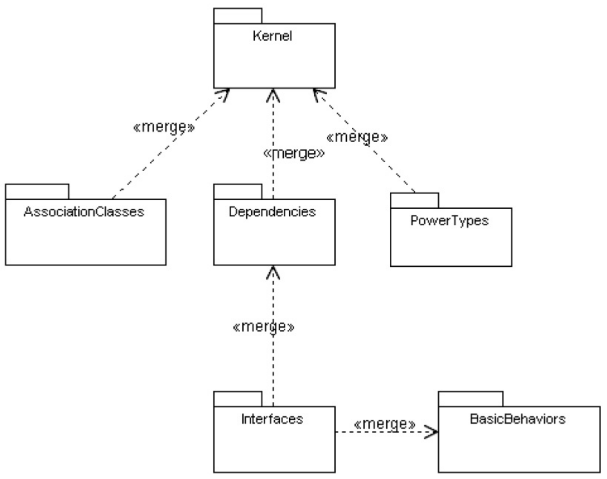


Figure 1: The metapackages and relationships for the metamodel of class diagrams in UML 2

tant, algebraic properties. One of the main results of the paper is that package merge as currently defined is not commutative, a property which appears to have been assumed to hold when package merge was used for the definition of the UML metamodel. Moreover, we present a semantically cleaner version of package merge based on subtyping, and also discuss its formalization and analysis with Alloy.

The rest of this paper is organized as follows. In Section 2, we discuss package merge in more detail, and provide background about UML and the UML metamodel. In Section 3, we explain how we used Alloy to model package merge. Section 4 details the results of our analysis. Section 5 provides a summary and conclusion.

2. BACKGROUND

2.1 The UML Metamodel

UML is defined by a metamodel — a model of the modeling language. The modeling language used to define UML is called the Meta-Object Facility (MOF) [15]. MOF is essentially a subset of UML itself. It includes concepts commonly found on UML class diagrams, such as classes (which can have superclasses, attributes and operations), associations, packages and package merge itself. To distinguish between a metamodel and a regular UML class diagram, we often preface element types by "meta" (for example, the *metaclass* State in a state machine metamodel versus the *class* Employee in a class diagram).

The UML metamodel is organized into packages; the packages are divided according to UML diagram type, and level of difficulty. For example, Figure 1 shows the packages that make up the metamodel for class diagrams. The *Kernel* package describes those concepts most commonly found in basic class diagrams; the other packages (such as *AssociationClasses* and *Interfaces*) describe more advanced class diagram modeling concepts.

2.2 Package Merge in Detail

Figure 2 shows an example of package merge. The package that is the source of the package merge relationship (in this case, the *BasicEmployee* package) is called the *receiving* package. The receiving package is the package whose contents are augmented by the merge. The *EmployeeLocation* package is called the *merged* package; this is the package that contains the additional elements that are merged into the receiving package. The outcome of merging the merged package into the receiving package is called the *resulting* package (shown in Figure 3). It is important to note that the resulting package is *implied* by the package merge relationship. This is similar to the way that inheritance is typically represented; when an inheritance relationship exists between two classes in a model, the child class is understood to contain all of the features that it inherits from its parent, as well as those features defined explicitly on the child. For instance, in Figure 2, the *BasicEmployee* package actually contains the contents of the resulting package from Figure 3.

We now describe the merge procedure in more detail. In the UML specification, package merge is defined by match rules, constraints and transformations (the merge rules). The merge rules are grouped according to element types (for instance, the merge rules for associations differ from the merge rules for operations). We can distinguish between the following three groups:

- *Match rules* define equality; if two elements in the merged and receiving packages match, they are considered to represent the same element, and are merged together to form one element in the resulting package; for example, in Figure 2, the class *Employee* in the *BasicEmployee* package matches the class *Employee* in the *EmployeeLocation* package, and so these two classes are recursively merged to produce the resulting class. In general, two elements match if they have the same name. Some exceptions include operations, which much have the same signature as well as the same name, and associations, which must have matching association end types. Elements from the merged package that do not have a match in the receiving package are simply copied to the receiving package.

- *Constraints* define preconditions on the merge; if a pair of receiving/merged packages violates any of the constraints, we say that the merge is *invalid*.

- *Transformations* define the postconditions of the merge; they describe how to resolve any conflicts when two matching elements are recursively merged (for example, if two matching classes have different attributes, or two matching attributes have different multiplicities).

In UML 2, package merge rules are defined only for those element types commonly found on class diagrams (for instance, classes, packages, associations, operations, and so on). This means that it is suitable for use on the UML metamodel. However, in UML, a package may contain other diagram types, such as state machine diagrams, or use case diagrams; these cannot currently be merged together and the extension of package merge to these types remains an open question.

The alternate version of package merge proposed in this paper is based on subtyping semantics. The merge rules for this version are defined such that the resulting package is a subtype of the receiving package.

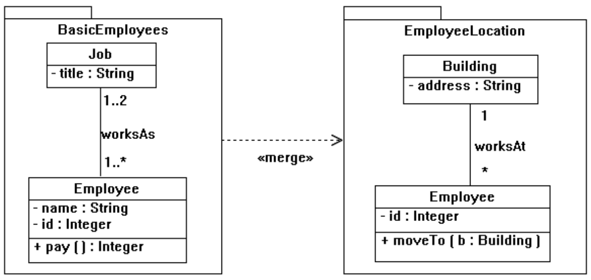


Figure 2: An example of package merge

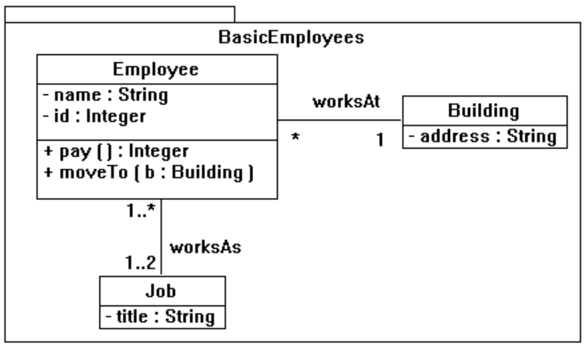


Figure 3: The resulting package of the merge in Figure 2

3. MODELING PACKAGE MERGE

Due to space constraints, only excerpts of the formalization can be described. For more details, we refer the reader to [23].

The Alloy model of package merge is divided into two modules: 1) the `Metamodel` module, which models the UML metamodel and 2) the `PackageMerge` module, which models the operation of package merge. There are actually two versions of the `PackageMerge` module (`OriginalPackageMerge` and `SubtypingPackageMerge`). The `OriginalPackageMerge` module models package merge as it is described in the UML standard; the `SubtypingPackageMerge` module models the alternate version of package merge that we have proposed based on subtyping semantics. Both `PackageMerge` modules use the signatures, predicates and functions defined in the `Metamodel` module.

3.1 Modeling the UML Metamodel

As mentioned in Section 2, package merge rules are currently only defined for elements commonly found on class diagrams; therefore, to model package merge, we only need to model the metapackage *Kernel*, rather than the entire UML metamodel. The UML metamodel maps well onto Alloy. The metaclasses and meta-attributes correspond to signatures and their fields; meta-associations are also modeled as fields. The multiplicities 1, 0..1, 1..* and 0..* map onto the Alloy keywords `one`, `lone`, `some` and `set` respectively. Additional semantic constraints on the metamodel are modeled as facts. Most of these constraints are written using OCL (the Object Constraint Language), which uses sets and predicate logic, and can also be fairly easily translated into Alloy.

Alloy suffers from poor performance when analyzing models with many signatures and fields; with 20 signatures or so, analysis is limited to a scope of 5-10 [11] (for our work, even smaller scope sizes had to be chosen). The subset of the UML metamodel that we are concerned with contains upwards of 30 metaclasses (which would each by modeled as a signature). Modeling package merge is very complex already, so it is desirable to reduce the number of signatures and fields in the `Metamodel` module as much as possible. On the other hand, if we remove too much detail, our analysis becomes uninteresting (for example, if we were to model classes, but not attributes or operations). We applied the following strategies to reduce the size and complexity of our Alloy model, while at the same time ensuring that our analysis would still produce meaningful results:

1. Collapsing the inheritance hierarchy: The UML metamodel has a fairly deep inheritance hierarchy; for example, the metaclass *Class* has no fewer than 6 ancestors. We find that, in most cases, we can simply collapse the hierarchy and put all inherited fields directly in the child class, especially in the case of abstract classes that have only one or two descendents. We keep only the abstract superclasses *Element*, *PackageableElement*, *TypedElement* and *MultiplicityElement*, which are used to allow polymorphism and simplify the model.

2. Removing similar classes: The metaclasses *DataType*, *PrimitiveType* and *Enumeration* are all very similar to *Class* - they all have names, attributes, operations and superclasses. We therefore choose to model only *Class*, and assume that any results apply to the other classes as well.

3. Not modeling uninteresting classes: The metaclasses *PackageImport*, *PackageMerge*, *ElementImport*, *Comment* and *Constraint* do not have any constraints or transformations associated with them; they are all copied into the receiving package as-is as the result of a merge. Since this is the least interesting outcome of a merge, we do not include these metaclasses in the model.

4. Merging multiple inheritance: MOF (and therefore the UML metamodel) allows multiple inheritance. An Alloy signature cannot extend more than one other signature; it is possible to model multiple inheritance, but only by defining it explicitly [11, Page 94]. However, by collapsing the inheritance hierarchy, we have already removed most instances of multiple inheritance. The one place that we would like to model multiple inheritance is for the metaclasses *Property* and *Parameter*, which both inherit from *MultiplicityElement* and *TypedElement* (*Property* is the metaclass for both class attributes and association ends). We do not want to collapse the inheritance hierarchy in this case since it would mean duplicating several fields in both subsignatures. In our model, since there are no classes that inherit from just one or the other, it makes sense to combine the two parent classes into one signature, called `TypedMultiplicityElement`.

5. Eliminating recursion: Alloy does not allow recursive calls. This makes it very difficult to model the merging

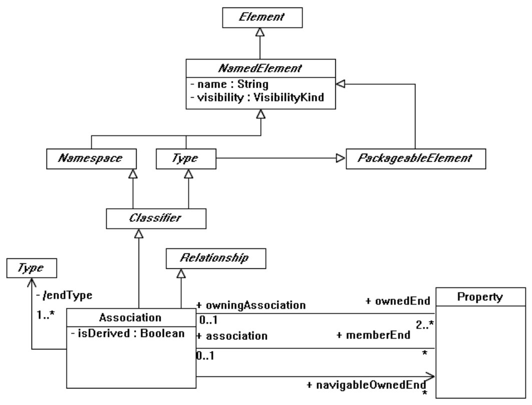


Figure 4: Part of the UML 2 metamodel for associations

of nested classes (a class that is contained by another class) or nested packages (a package that is contained in another package). However, since nested classes and packages are merged in the same way as non-nested ones, we can simply not model them without affecting our results.

6. Not modeling any derived attributes or associations: The UML metamodel includes many derived attributes and associations, which are values that can be calculated based on other information in the model — for example, the *isUnique* attribute of an operation is derived from its return result. We do not include any of these derived values as fields in our Alloy signatures, since the number of fields also negatively affects the performance of the Analyzer. Instead, these values are calculated on-the-fly by functions or predicates if they are required.

The UML metamodel also makes use of the primitive types string, integer and boolean. The only one of these that is natively supported by the Alloy language is integer. To model boolean values, we import the predefined module `boolean` that comes with Alloy. This module includes constants for true and false, along with functions for logical operations such as "and" and "or". To model strings (which are used for element names), we add a signature `String` with no fields. It is not necessary to model any more detail about strings, such as defining a string as a list of characters, or defining any string functions, like concatenation. For the purposes of package merge, we only need to be able to compare two names to see whether or not they are the same one — if the `name` fields of two signatures refer to the same `String` instance, then they are equal.

As an example of converting the UML metamodel to an Alloy model, consider Figure 4, which shows a part of the metamodel for associations in the UML specification. Figure 5 shows the Alloy signature for associations. The class *Association* is modeled as a signature that extends the signature `PackageableElement` (since we have collapsed its other superclasses). The associations between *Association* and *Property* are modeled as fields of the signature `Association` (lines 4-6), as is the attribute *isDerived* (line 3). We do not include the derived association *endTypes* in the model at all. We model some multiplicities through the field declarations, but since Alloy does not have a multiplicity marking for 2..*, we include it as a set, and constrain it further in an attached fact (line 9). In the attached fact, we also model some additional semantic constraints from the association metamodel (see Figure 6). For instance, line 10 corresponds to constraint 4, while line 11 corresponds to constraint 5.

```
1  sig Association extends PackageableElement
2  {
3    isDerived: Bool,
4    memberEnds: set Property,
5    ownedEnds: set Property,
6    navigableOwnedEnds: set Property
7  }{
8    ownedEnds in memberEnds &&
         navigableOwnedEnds in ownedEnds
9    #memberEnds >= 2
10   (some end:memberEnds | end.isComposite =
         True) => #memberEnds = 2
11   #memberEnds >2 => memberEnds = ownedEnds
12   all e1, e2:memberEnds | e1.@name = e2.
         @name => e1=e2
13 }
```

Figure 5: The signature `Association`

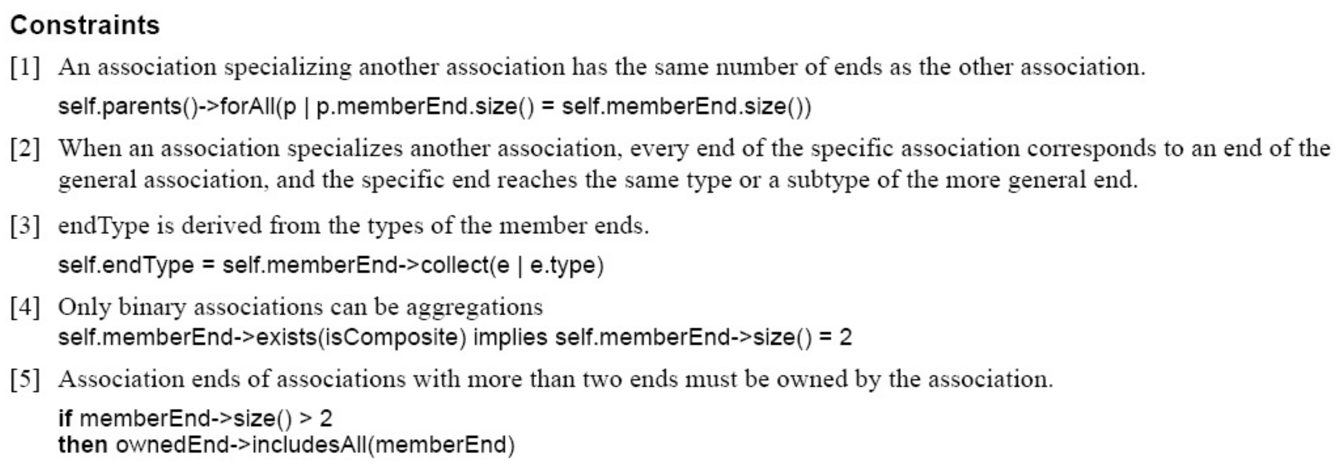
The `Metamodel` module also includes some useful functions and predicates for working with the metamodel. The most important of these is the predicate `equals`. As shown below, `equals` takes two `Elements` (which is the parent of all the other signatures) as parameters, and returns true if and only if they are equivalent:

```
pred equals(e1:Element, e2:Element)
{
  (e1 in Class && e2 in Class &&
      classEquals(e1,e2)) ||
  (e1 in Property && e2 in Property &&
      propertyEquals(e1,e2)) ||
  (e1 in Association && e2 in Association
     && associationEquals(e1,e2)) ||
  (e1 in Operation && e2 in Operation &&
      operationEquals(e1,e2)) ||
  (e1 in Parameter && e2 in Parameter &&
      parameterEquals(e1,e2))
}
```

where Predicate `classEquals`, for instance, will return true iff both classes have the same name, equivalent sets of attributes and operations, and the same superclass. Equivalence on properties, associations, operations, and parameters is defined analogously. Predicate `equals` is used by both `PackageMerge` modules.

3.2 Modeling the Operation of Package Merge

The `OriginalPackageMerge` and `SubtypingPackageMerge` modules contain mostly the exact same predicates, with only a few small differences. We will describe a generic `PackageMerge` module, and note any differences as they arise. We model package merge as a 3-ary predicate. The predicate `mergePackage` takes three packages as parameters (the receiving, merged and resulting packages), and returns true if and only if the resulting package parameter is the result of


Constraints

[1] An association specializing another association has the same number of ends as the other association.

self.parents()->forAll(p | p.memberEnd.size() = self.memberEnd.size())

[2] When an association specializes another association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end.

[3] endType is derived from the types of the member ends.

self.endType = self.memberEnd->collect(e | e.type)

[4] Only binary associations can be aggregations
self.memberEnd->exists(isComposite) implies self.memberEnd->size() = 2

[5] Association ends of associations with more than two ends must be owned by the association.

if memberEnd->size() > 2
then ownedEnd->includesAll(memberEnd)

Figure 6: The metamodel constraints on associations written in English and OCL

merging the contents of the receiving and merged package parameters. Figure 7 shows this predicate. The predicate returns true if the following conditions are met:

1. The merge must be valid (line 3) (all constraints of package merge must be satisfied — the predicate `mergeIsValid` is described in more detail below).
2. For every owned member of the merged package, if there is no matching element in the owned members of the receiving package, there must be an owned member of the resulting package that is equal to it (i.e., a copy of it), and vice-versa for the receiving package (lines 4-7).
3. If a merged and receiving element are equal, there must be an element equal to them in the resulting package (lines 8-9).
4. For all the associations and classes contained in the merged package with a matching class or association in the receiving package, there is a class or association in the resulting package that is the result of merging the merged and receiving elements (lines 10-15). The functions `ownedClasses` and `ownedAssociations` are defined in the `Metamodel` module, and return the set of classes and associations, respectively, owned by the package passed as a parameter.
5. The resulting package must contain only those classes and associations that are the result of merging together the merged and receiving packages. Without this condition, there would be nothing to prevent the resulting package from having additional contents that do not come from either package (lines 17-18).

There is a merge predicate `mergeXXX` for every signature in the Alloy metamodel (the `mergePackage` predicate calls the predicates `mergeClass` and `mergeAssociation` in particular). These predicates model the transformation rules. All of these predicates have a form similar to `mergePackage`: the elements that are contained by the element type are merged according to conditions 2 through 5 above (elements are contained if they are related by a composite association), and any additional features are merged according to some transformation. The predicates bottom out with `mergeParameter`

```
1  pred mergePackage(receiving:Package,merged:
       Package,resulting:Package)
2  {
3    mergeIsValid(receiving, merged)
4    all e1:receiving.ownedMembers | (no e2:
         merged.ownedMembers |
5      matches(e1, e2)) => some e3:resulting.
           ownedMembers | equals(e1, e3)
6    all e1:merged.ownedMembers | (no e2:
         receiving.ownedMembers |
7      matches(e1, e2)) => some e3:resulting.
           ownedMembers | equals(e1, e3)
8    all e1:receiving.ownedMembers | (some e2:
         merged.ownedMembers |
9      equals(e1, e2)) => some e3:resulting.
           ownedMembers | equals(e1, e3)
10   all a1:ownedAssociations(receiving), a2:
         ownedAssociations(merged) |
11     matches(a1, a2) => (some a3:
           ownedAssociations(resulting) |
12     mergeAssociation(a1, a2, a3))
13   all c1:ownedClasses(receiving), c2:
         ownedClasses(merged) |
14     matches(c1, c2) =>
15     (some c3:ownedClasses(resulting) |
           mergeClass(c1, c2, c3))
16
17   all e1:resulting.ownedMembers |
18     some e2:merged.ownedMembers + receiving
           .ownedMembers | matches(e1, e2)
19 }
```

Figure 7: The `mergePackage` predicate

and `mergeProperty`, since properties and parameters do not own other elements.

The `mergeXXX` predicates make use of the `equals` predicate, which is defined in the `Metamodel` module, and the `matches` predicate. Like `equals`, `matches` takes two `Elements` as parameters, and returns true if the two elements match according to the match rules for their type. For example, the `associationMatches` predicate looks like:

```
pred associationMatches(a1:Association, a2:
    Association)
{
  a1.name = a2.name
  #a1.memberEnds = #a2.memberEnds
  all e1:a1.memberEnds | some e2:a2.
      memberEnds | propertyMatches(e1, e2)
}
```

This predicate returns true if both associations have the same name, the same number of ends, and if these ends match.

We model the checking of constraints with the predicate `mergeIsValid`, which takes the receiving package and merged package as parameters, and returns true if and only if the merge of these two packages violates no constraints. The `mergeIsValid` predicate calls the predicates `classConstraints` and `associationConstraints` on matching pairs of classes and associations in the receiving and merged packages:

```
pred mergeIsValid(receiving:Package, merged
    :Package)
{
  all e1:ownedClasses(receiving), e2:
      ownedClasses(receiving) |
    matches(e1, e2) => classConstraints(e1,
         e2)
  all e1:ownedAssociations(receiving), e2:
      ownedAssociations(receiving) |
    matches(e1, e2) =>
        associationConstraints(e1, e2)
}
```

The `classConstraints` and `associationConstraints` predicates in turn call predicates that check constraints for operations, parameters and properties. The constraint checking predicates also bottom out with parameters and properties. For example, the predicate `propertyConstraints` looks like:

```
pred propertyConstraints(receiving:Property
    , merged:Property)
{
  conforms(receiving.type, merged.type)
  isTrue(merged.isComposite) => isTrue(
      receiving.isComposite)
}
```

This predicate checks the property constraints for the original version of package merge (the types of the properties must be conforming, and the receiving property must be a composite if the merged property is a composite); in the subtyping version of package merge, the constraint regarding composite properties is removed.

4. ANALYSIS AND RESULTS

A central goal of our analysis was to check for algebraic and semantic properties of package merge. We can think of package merge as an operation that takes two inputs (the receiving and merged packages) and returns the resulting package. Using the model of package merge, we can check to see if certain algebraic properties of this operation hold (e.g., associativity, commutativity, idempotency). In addition to being useful information for modelers using package merge, some of these properties are *necessary* properties for package merge to have. If package merge lacks these necessary properties, it indicates that the merge rules must be changed to ensure them. Although the UML specification appears to have assumed some of these properties, it does not articulate any.

We also used Alloy to analyze the semantics of package merge. We formalized our definition of package subtyping in Alloy, and used it to check that the subtyping version of package merge does indeed ensure that the resulting package is a subtype of the receiving package.

4.1 Properties Analyzed

We checked the properties of package merge by writing assertions to represent each, and using the Analyzer to see if the assertions held. We checked for the following properties:

- *Uniqueness*: Uniqueness means that if package A is merged into package B, there is only one possible resulting package (one outcome of the merge). This is a necessary property for package merge to have; if it does not hold, it indicates that there is some ambiguity in the definition of the merge rules. Checking for this property helps us to make sure that the package merge operation has been completely defined, and that there are no missing rules. Additionally, it can help us find problems with our formal model — since this is a property we expect package merge to have, any counterexamples will likely indicate a problem with the model, rather than package merge itself. This property can be formalized in Alloy as:

```
assert mergeIsUnique
{
  all receiving, merged, resulting1,
      resulting2:Package |
    mergePackage(receiving, merged,
        resulting1)
   && mergePackage(receiving, merged,
        resulting2) =>
    packageEquals(resulting1,
        resulting2)
}
```

- *Associativity*: In general, a binary function f is associative if $f(x, f(y,z)) = f((x,y),z)$ for all x, y, and z. This is an important property for package merge to have. Merges can be chained, that is, a merged package can also be a receiving package. For instance, package A may merge in package B which also merges in C (see, e.g., Figure 1). The graphical notation for package merge chosen in UML does not allow for an evaluation order to be expressed. Therefore, it appears that the designers of package merge intended package merge to be associative. Assuming that merge is implemented by a binary function, the associativity property can be expressed in predicate logic as follows:

$$\forall receiving, merged1, merged2 : Package. \\ merge(receiving, merge(merged1, merged2)) = \\ merge(merge(receiving, merged1), merged2))$$

In Alloy, we captured it by:

```
assert mergeIsAssociative
{
  all receiving , merged1 , merged2 ,
      intermediate , resulting : Package
        |
    mergePackage(receiving , merged1 ,
        intermediate1) &&
    mergePackage(intermediate1 , merged2
        , resulting) &&
    mergePackage(merged1 , merged2 ,
        intermediate2) =>
      mergePackage(receiving ,
          intermediate2 , resulting)
}
```

- *Commutativity*: A binary function f is commutative, if $f(x, y) = f(y, x)$ for all x and y. However, since a receiving package can have merge relationships with more than one package (see, e.g., Figure 1), commutativity analyses to answer the following two questions appear appropriate: 1) Can the order of the merged packages be changed without changing the result of the merge? 2) Can the receiving package and one of the merged packaged be swapped without changing the result of the merge?

  1. *Commutativity between the merged packages:* Assuming that merge is a binary function, this property can be expressed in predicate logic as

     $$\forall receiving, merged1, merged2 : Package. \\ merge(merge(receiving, merged1), merged2) \\ = \\ merge(merge(receiving, merged2), merged1)$$

     In Alloy, we have it expressed by:

     ```
     assert mergeIsCommutative1
     {
       all receiving , merged1 , merged2 ,
           resulting , intermediate1 ,
       intermediate2: Package |
         mergePackage(receiving , merged1
             , intermediate1) &&
         mergePackage(intermediate1 ,
             merged2 , resulting)
         && mergePackage(receiving ,
             merged2 , intermediate2)  =>
         mergePackage(intermediate2 ,
             merged1 , resulting)
     }
     ```

     Again, this an important property for package merge to have, because the graphical notation does not allow the order in which the merged packages are to be merged into the receiving package to be expressed. Again, this property appears to have been assumed by the designers of merge. If package merge is not allow for packages to be merged in in an arbitrary order, then either the definition will have to be changed, or some notation for indicating merge order will need to be defined.

  2. *Commutativity between receiving and merged package:* Assuming that merge is a binary function the property can be expressed in predicate logic by:

     $$\forall receiving, merged : Package. \\ merge(receiving, merged) = \\ merge(merged, receiving)$$

     Unlike uniqueness, associativity, and the commutativity between merged packages, commutativity between the receiving and the merged package is not a necessary property for package merge. In fact, it is easy to see that package merge cannot be completely commutative — if we merge package A into package B, then the resulting package will have the same name as A; if we perform the merge in the other direction, the resulting package will have the same name as B. However, we can check for a less strict version of commutativity by comparing only the contents of the resulting packages. Since the `mergePackage` predicate does ignore the name of the package and only checks that the contents of the resulting package are the result of the merge of the receiving and merged packages, we can write the Alloy assertion as:

     ```
     assert mergeIsCommutative2
     {
       all receiving , merged , resulting:
           Package |
         mergePackage(receiving , merged ,
             resulting) =>
         mergePackage(merged , receiving ,
             resulting)
     }
     ```

- Idempotency: A unary function f is idempotent if it can be applied multiple times to the same value without changing the result, that is, if $f(f(x)) = f(x)$ for all x. Although package merge is not unary, we can check for a similar property; if package A is merged into package B, and then package A is merged again into the resulting package, does the result change? Assuming that merge is a binary function, the property is expressed in predicate logic as:

  $$\forall receiving, merged : Package. \\ merge(merge(receiving, merged), merged) = \\ merge(receiving, merged)$$

  In our Alloy formalization, it is captured by

  ```
  assert mergeIsIdempotent
  {
    all receiving , merged , resulting :
        Package |
      mergePackage(receiving , merged ,
          resulting)
  ```

```
    => mergePackage(resulting, merged,
         resulting)
}
```

Intuitively, we expect this property to hold. Merging in a package a second time should have no effect.

- *Subtyping*: This is a property that can only be checked for the subtyping version of package merge. Unlike the other properties, it applies to package merge *semantics*. We modeled our definition of package subtyping in Alloy, and used this model to see if our revised definition of package merge does indeed ensure that the resulting package is a subtype of the receiving package:

```
assert mergeEnsuresSubtyping
{
  all receiving, merged, resulting:
       Package |
     mergePackage(receiving, merged,
         resulting) =>
     isSubtype(resulting, receiving)
}
```

4.2 Results of the Analysis

The results of the analysis of both versions of package merge are summarized in Table 1. The analyses were performed on a machine with 3 GB of RAM, a 3.4 GHz processor and a Windows XP Professional platform, using version 3.0 beta of the Alloy Analyzer. A default scope of 3 was used for all signatures (since larger scopes proved to consume too many resources), except for the signature `Element`. Since `Element` is the parent signature of all the other metamodel signatures (e.g., `Class`, `Association`, etc.), limiting its scope to 3 would have limited the total number of instances of its child signatures to 3 as well. We therefore set the scope of `Element` separately (as shown in Table 1). The table of results also shows the time that it took to run each assertion check.

Our analysis showed that the original version of package merge is unique, idempotent, and associative, but is not commutative in neither of the two senses. The subtyping version, however, is idempotent, unique, associative and commutative in both senses. The analysis also shows that the subtyping version of package merge does indeed ensure subtyping.

The failure of commutativity between the receiving and the merged package, and, more surprisingly, commutativity between the merged packages is due to the fact that some of the constraints of original package merge are not symmetric. Figures 8 and 9 show the counterexamples that are returned by the Alloy Analyzer when it checks the assertions `mergeIsCommutative1` and `mergeIsCommutative2` on the `OriginalPackageMerge` module (these examples have been redrawn as class diagrams for readability, rather than the graphs returned by Alloy). For example, the merge in Figure 9 is not commutative, because the merge is valid in one direction, but not valid in the other (because of the constraint that the receiving property must be composite if the merged is composite — or, as formalized in Alloy:

```
isTrue(merged.isComposite)=>
   isTrue(receiving.isComposite)
```

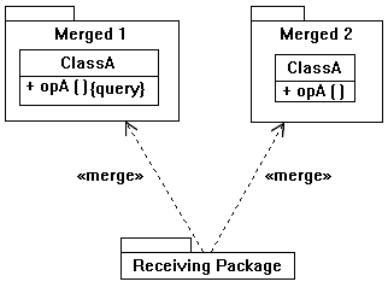


Figure 8: A counterexample for the commutativity between the merged packages

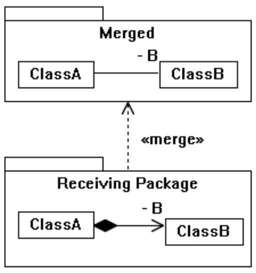


Figure 9: A counterexample for the commutativity between the receiving and merged package

A similar problem impedes commutativity between the merged packages, as shown in Figure 8. If package *Merged 1* is merged into the receiving package first, there is no problem; however, if package *Merged 2* is merged in first, then the other merge becomes invalid (due to a constraint on the *isQuery* property of operations), and so the order of merging becomes important. As mentioned in Section 4.1, this is particularly worrisome, since there is no notation for denoting merge order. The problem is that the constraints themselves are not commutative; if the merging A into B is valid, merging B into A might not be.

Indeed, we found that the original version of package merge *is* commutative in both senses if we eliminated the problem of invalid merges; i.e., it is commutative in both senses in all cases where the merges do not violate any constraints. For example, the following two assertions hold, as shown in Table 1:

```
assert validMergeIsCommutative1
{
  all receiving, merged1, merged2,
       resulting, intermediate1,
     intermediate2: Package | mergePackage(
         receiving, merged1, intermediate1)
     && mergePackage(intermediate1, merged2,
           resulting)
     && (mergePackage(receiving, merged2,
         intermediate2)
     && mergeIsValid(intermediate2, merged1)
         )
     => mergePackage(intermediate2, merged1,
           resulting)
}
```

Table 1: Rensults of the Alloy Analysis (hrs:mins:secs)

Property	Scope	Original version	Subtyping version
Ensures subtyping	3 (6 `Element`)	N/A	Yes (00:06:15)
Uniqueness	3 (6 `Element`)	Yes (00:06:54)	Yes (00:08:05)
Associativity	3 (5 `Element`)	Yes (00:16:54)	Yes (00:17:23)
Commutativity 1	3 (5 `Element`)	No (00:05:04)	Yes (00:13:04)
Commutativity 1 assuming validity	3 (5 `Element`)	Yes (00:13:11)	Yes (N/A)
Commutativity 2	3 (6 `Element`)	No (00:10:02)	Yes (01:53:43)
Commutativity 2 assuming validity	3 (6 `Element`)	Yes (01:43:21)	Yes (N/A)
Idempotency	3 (6 `Element`)	Yes (00:54:38)	Yes (01:00:39)

```
assert validMergeIsCommutative2
{
  all receiving, merged, resulting:Package
      |
    (mergeIsValid(receiving, merged)
    && mergeIsValid(merged, receiving)
    && mergePackage(receiving, merged,
         resulting))
    => mergePackage(merged, receiving,
         resulting)
}
```

The assertion `validMergeIsCommutative2` checks that merge is commutative if it is valid in both directions; `validMergeIsCommutative1` checks that the merged packages can be commuted, if the merge is valid no matter which package is merged in first. We did not need to check these properties for the subtyping version of package merge; since we have already found that it is commutative in all cases, it is therefore also commutative assuming valid merges.

4.3 Related Work

A related paper describes the failure of backwards compatibility of package merge and attempts to classify model extension mechanisms in very general terms [24]. It does not discuss the subtyping version, any of the properties, or the formalization and analysis of the two versions using Alloy.

Although package merge is new to UML 2, similar mechanisms exist. For instance, package extension mechanisms are part of the Meta-Modeling Framework, a framework for formal metamodeling [7] (which, to an extend, has been subsumed by Xactium's XMF-Mosaic [22]), the Visual and Precise Metamodeling framework [21], the Atlas Model Weaver [4], and the Glue Generator tool [6]. The Epsilon Merging Language [12] allows the description of very wide class of model merge operations. The Catalysis method for component-based modeling using UML [8] also allows the contents of two packages to be combined when one package imports another. Package merge also has many similarities to the operation of *composition* in aspect-oriented modeling [1, 9]; the functional components of a system are modeled separately from its aspects, and these two pieces must be composed to form a complete view. All of these mechanisms (including package merge) can be thought of as specific instances of *model merge*, which is one of the operators in the generic model management algebra proposed by Bernstein [3, 2].

The notion of package subtyping is based on work done by Steel and Jézéquel about model types [19, 20]. They define the type of a model as the collection of types in its metamodel; since a package can also be thought of as a collection of its contained elements, a similar definition of subtyping can apply to models and packages.

Other researchers have used Alloy to overcome UML's lack of precision and reasoning capabilities. For instance, [18, 13] present alternative definitions of the UML metamodel and use Alloy for analysis. In [5], on the other hand, Alloy's analysis is leveraged by defining a transformation from the UML metamodel into a metamodel of Alloy. Just like in our work, all these approaches use Alloy to formalize (different versions of) the UML metamodel. However, none of them use the encoding to analyze package merge.

5. SUMMARY AND CONCLUSION

We have presented an Alloy formalization of two versions of UML package merge, and a list of properties. The Alloy analysis revealed that the original version of package merge has more problems than the ones already reported in [24]. To conclude, we agree that package merge is "an advanced modeling feature intended primarily for metamodel builders" [17].

We also conclude that the current version is not yet ready either for use by UML users in general (rather than just metamodel builders), or to be extended to other types of model elements such as interactions or state machines. Rather, it should be corrected. A conservative correction is discussed in [23]. A more drastic solution would be to adopt a formulation of package merge based on subtyping which avoids the problems of the original formulation and appears to provide a promising basis for extending package merge to other types of model elements.

Alloy allowed us to formalize the two merge operations and a number of properties in a precise, succinct, and elegant manner. Its analysis was very useful for debugging the model and gave us considerably more confidence in its correctness. Moreover, it revealed a couple of surprising property violations. The produced counter examples allowed us to track down the reason of the violations quickly. On the downside, modifications to the model were required to make its analysis tractable and to account for the lack of multiple inheritance. Even with these modifications, the tractable scopes were still quite small; more comprehensive analyses are left for future work.

There is currently a considerable interest in academia and industry in modeling languages. For instance, the UML and MDA efforts led by the OMG, and the numerous conferences and workshops devoted to the topic bear witness to this. Moreover, industrial tools supporting the work with models and sophisticated operations on them are beginning to be available. For instance, IBM's Rational Software Architect provides support for comparison, merging, and team development of models [10]. As our experience with package merge in UML 2 shows, not all of this work is backed up by thorough research to an desirable extend. Alloy proved to be a very useful vehicle for our investigations. We suggest that advanced modeling and analysis tools such as theorem provers and constraint checkers (such as Alloy) be used to pave the way towards a more effective theory and practise of modeling language design in general and of model merge in particular.

6. ACKNOWLEDGEMENTS

We would like to thank Bran Selic and Jim Amsden for taking the time to answer our questions about package merge. Thanks also go to the anonymous referees for their helpful comments.

7. REFERENCES

[1] B. Baudry, F. Fleurey, R. France, and R. Reddy. Exploring the relationship between model composition and model transformation. In *International Workshop on Aspect-Oriented Modeling (AOM'05)*, 2005.

[2] P. Bernstein. Applying model management to classical meta data problems. In *Conference on Innovate Database Research (CIDR'03)*, 2003.

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

[4] J. Bézivin, F. Jouault, and D. Touzet. An introduction to the atlas model management architecture. Technical Report 05-01, Research Report Laboratoire D'Informatique de Nantes Atlantique (LINA), 2005.

[5] B. Bordbar and K. Anastasakis. Uml2alloy: A tool for lightweight modelling of discrete event systems. In *IADIS International Conference in Applied Computing*, Algarve, Portugal, 2005.

[6] S. Bouzitouna, M. P. Gervais, and X. Blanc. Model reuse in mda. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP05)*, Las Vegas, USA, June 2005.

[7] T. Clark, A. Evans, and S. Kent. Engineering Modelling Languages: A Precise Metamodelling Approach. In H. Weber, editor, *Fundamental Approaches to Software Engineering (FASE'02)*, LNCS 2306, pages 159–173, Grenoble, France, April 2002.

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

[9] G. Georg, R. France, and I. Ray. Composing aspect models. In *4th Workshop on Aspect-Oriented Software Development Modeling With UML*, 2003.

[10] IBM Rational Software. Rational Softare Architect: Product overview. `www-306.ibm.com/software/awdtools/architect/swarchitect`.

[11] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.

[12] D. Kolovos, R. Paige, and F. Polack. Merging models with the Epsilon merging language. In *Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, LNCS, Genova, Italy, October 2006. Springer Verlag.

[13] A. Naumenko and A. Wegmann. A metamodel for the unified modeling language. In S. C. E. J.-M. Jzquel, H. Hussmann, editor, *5th International Conference on the Unified Modeling Language: Model Engineering, Concepts, and Tools (UML 2002)*, LNCS 2460, pages 2–17, Dresden, Germany, September/October 2002. Springer Verlag.

[14] Object Management Group. *Unified Modeling Language: Superstructure (version 2.0, formal/05-07-04)*, August 2005.

[15] Object Management Group. *Meta Object Facility (MOF) Core Specification (version 2.0, formal/06-01-01)*, January 2006.

[16] Object Management Group. *Unified Modeling Language: Infrastructure (version 2.0, formal/05-07-05)*, March 2006.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Object Technology Series. Addison-Wesley, 2005.

[18] A. Simons and C. Fernandez. Using alloy to model-check visual design notations. In *Sixth Mexican International Conference on Computer Science*, pages 121–128, 2005.

[19] J. Steel and J.-M. Jézéquel. Typing relationships in MDA. In D. Akehurst, editor, *Second European Workshop on Model-Driven Architecture (EWMDA-2)*, 2004.

[20] J. Steel and J.-M. Jézéquel. Model typing for improving reuse in model-driven engineering. In L. Briand and C. Williams, editors, *MODELS/UML'2005*, 2005.

[21] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[22] Xactium Ltd. `www.xactium.com`.

[23] A. Zito. UML's package extension mechanism: Taking a closer look at package merge. Master's thesis, School of Computing, Queen's University, September 2006.

[24] A. Zito, Z. Diskin, and J. Dingel. Package merge in UML 2: Practice vs. theory? In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS/UML 2006*, October 2006.