Improving Naming and Grouping in UML

Antonio Vallecillo

GISUM/Atenea Research Group, Universidad de Málaga, Spain
av@lcc.uma.es

Abstract. The package is one of the basic UML concepts. It is used both to group model elements and to provide a namescope for its members. However, combining these two tasks into a single UML concept can become not only too restrictive but also a source of subtle problems. This paper presents some improvements to the current UML naming and grouping schemata, using the ideas proposed in the reference model of Open Distributed Processing (ODP). The extensions try to maintain backwards compatibility with the existing UML concepts, while allowing more flexible grouping and naming mechanisms.

1 Introduction

The UML [1] is probably the most widely used modeling notation nowadays. It is currently applied for the specification of many different kinds of systems and is supported by a wide range of tools. However, UML is far from being a perfect notation. It has grown too much to accommodate too many concepts and languages. The need for backwards compatibility with previous versions, and the OMG's design-by-committee approach to standardization have not helped much either. This has resulted in a large, complex and brittle metamodel for UML, which represents a challenge for all its stakeholders—in particular for its users and for the UML tool builders. More importantly, some of its basic concepts were probably developed with some restricted usages in mind. The problem is that they are now challenged by the new MDE requirements, which are stretching these concepts beyond their original intent.

In this paper we are concerned with the limitations of one of the basic UML concepts, the Package, which is used in UML to group model elements and to provide a namescope for its members. Although combining these two goals into a single UML concept can be appropriate in some cases, in general its proves to be too restrictive for organizing model elements in a flexible way because it ties elements with names, and names with packages. Thus, the UML grouping mechanism only allows non-overlapping classifications of model elements, and there is no easy way to implement federated modeling [2]. Furthermore, the two operations associated to these objectives (PackageMerge and PackageImport) become jeopardized by the limitations of UML packaging. For instance, name resolution does not work well under the presence of PackageImport, something critical in the case of OCL expressions and constraints [3]. Several semantic problems have also been reported for PackageMerge in [4,5].

This paper proposes some improvements of the current UML naming and grouping mechanisms, trying to respect backwards compatibility with the existing concepts. Import and merge operations will also be re-defined using the new concepts presented in

R.B. France et al. (Eds.): ECMFA 2011, LNCS 6698, pp. 345–360, 2011.

this paper. Our proposal is based on existing international standards of Open Distributed Processing (ODP) [6,7]. ODP offers a set of mature and well-defined concepts and solutions for coping with the inherent complexity of the specification of large distributed applications. In particular, we base our proposal on the ODP Naming standard [8], and on the organizational concepts defined in the ODP foundations [6, Part 2].

The structure of this paper is as follows. After this introduction, section 2 provides a summary of the Package element and the import and merge relationships defined in UML 2. Section 3 presents the problems that we have detected with them. Section 4 introduces our proposal and the concepts and mechanisms that we have defined to address these problems. Then, section 5 discusses how our proposal can extend the current UML notation. Finally, section 6 compares our work with other related proposals, and section 7 draws some conclusions.

2 The UML Package and Its Related Operations

The package is the mechanism available in UML for grouping modeling elements (including other packages) and it also provides a namespace for its members. A namespace is an element in a UML model that contains a set of elements that can be identified by name [1, clause 7.3.34]. The namespace is a named element itself. A package owns its members, with the implication that if a package is removed from a model, so are the elements owned by the package.

There are two package operations related to grouping and naming: PackageImport and PackageMerge. A package import is a relationship that identifies a package whose members are to be imported by a namespace. It allows the use of unqualified names to refer to package members from other namespaces [1, clause 7.3.39]. Owned and imported elements have a visibility that determines whether they are available outside the package.

Conceptually, a package import is equivalent to individually importing each member of the imported namespace using the ElementImport operation, which identifies an element in another package. This allows the element to be referenced using its name without any qualifier, as if they were owned elements. Elements defined in an enclosing namespace are also available using their unqualified names in the enclosed namespaces.

When importing an element it is possible to use an alias for it (to, e.g., avoid name clashes), and to decide whether the imported element can be further imported or not by other packages (the visibility of the imported element has to be either the same or more restricted than that of the referenced element).

Element importation works by reference, i.e., it is not possible to add features to the element import itself, although it is possible to modify the referenced element in the namespace from which it was imported [1, clause 7.3.15]. In case of name clashes with internal elements, the imported elements are not added to the importing namespace, and the name of those elements must be qualified in order to be used. In case of a name clash with an outer name in the importing namespace, the outer name is hidden, and the unqualified name refers to the imported element (the outer name can be accessed from this moment on using its qualified name). If more than one element with the same name is imported to a namespace as a consequence of multiple element or package imports, the elements are not added to the importing namespace.

A package merge defines how the contents of one package (the source) are extended by the contents of another package (the target) [1, clause 7.3.40], combining them together. Thus, the contents of the receiving package are (implicitly) modified and become the result of the combination of the two packages. In other words, the receiving package and its contents represent both the operand and the results of the package merge, depending on the context in which they are considered.

PackageMerge is normally used to integrate the features of model elements defined in different packages that have the same name and are intended to represent the same concept. It is extensively used in the definition of the UML metamodel, allowing an incremental definition of the metaclasses by which each concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end [1]. In terms of semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed.

The resulting model contains the duplicate-free union of the two merged packages, matching elements by names and types, and joining together all their features (properties, references, operations, constraints, etc.). In case of any kind of conflicts, the merge is considered ill-formed and the resulting model that contains it is invalid.

The result of combining the merge and import operations is also defined in UML. Importing a receiving package of a merge will import the combined contents of the merge (because the receiving package contains the result of the merge operation), while importing a merge package (i.e., the target of the merge) will only import its contents. In case of an element import owned by the receiving package of a merge, it will be transformed into a corresponding element import in the resulting package. Imported elements are not merged (unless there is also a package merge to the package owning the imported element or its alias).

3 Current Limitations of the UML Package

As we mentioned above, the fact that the same concept (the Package) provides at the same time the grouping and naming mechanisms for the UML language, together with the complex semantics of import and merge operations, may lead to some problems and restrictions. These are described in this section.

3.1 Ownership of Packaged Elements

In UML, a package owns its grouped elements [1]. This imposes a hierarchical decomposition of the model elements into a tree structure (not a graph), in which a model element can belong to at most one package.

Although the use of tree structures greatly simplifies the handling and grouping of model elements, it becomes a strong limitation if a modeler wants to group model elements according to different classifications, each one using a different criteria. For example, we could organize our model elements depending on whether they represent structural or behavioural features of the system being modeled (e.g., the software architecture in one package, the specification of the interactions in another). Another organization can divide the model elements depending on whether they represent internal elements of the system,

or part of the interfaces with other systems. We could also classify the model elements depending on their level of criticality, or the project phase they belong to. Forcing one organizational structure is not acceptable: normally there is no single classification that fits all conceptual dimensions in which modeling elements can be organized.

A proper grouping schema for organizing models should be able to allow modeling elements to belong, in principle, to more than one group.

3.2 Naming Schema

A UML package also provides a namespace for its members. In UML, a namespace is an element that owns a set of elements that can be identified by name.

The fact that UML elements may have at most one name can be a strong limitation in some contexts, for example in heterogeneous federated environments. In such cases there is no single global naming schema in which only one selected naming convention applies to all entities, and there is no single authority that administers all names. We cannot ignore the existing independently developed naming schemes already in use, or the ones currently being proposed.

Moreover, there are many situations in which the name of an element depends on the context in which it is used, so names cannot be fixed and absolute.

A proper naming schema should be able to accommodate these different naming schemata and allow their interconnection, while at the same time maintaining name unambiguity in each context of use.

3.3 Name Resolution

Another problem of the current UML naming schema has to do with name resolution, due to the semantics of ElementImport operation. The situation gets even worse under the presence of PackageImport and PackageMerge operations, as mentioned in [3].

This is a critical issue for those languages, such as OCL [9], that provide navigation facilities to refer to specific model elements. The environment of an OCL expression defines what model elements are visible and can be referred to from the expression. Such references are done by name, using either single names or package-qualified names. In principle the UML and OCL naming schemata are aligned, although it is not completely clear whether they are fully compatible or not, specially when import or merge operations are in place with their complex rules and with aliasing. MOF languages also have complex visibility rules, again aggravated by the importing relationships that can exist on a model. For example, the same package can have two distinct interpretations depending on its role as a source or as a target of a package merge or import relationship. The context-sensitive interpretation of a package has subtle implications for the name resolution of OCL constraints in the context of a package with two distinct interpretations about its extent [3].

There are other problematic situations that can lead to undesirable consequences, due to the potential side-effects of the import operation. For example, suppose a package P containing a constraint C that refers to a class A in an outer package. If P now imports another package Q that contains a class named A, the name resolution rules will hide the old class A and resolve name A to the newly imported class. This could easily break

the constraint or, even worse, inadvertently change its behaviour. Another undesirable situation can happen if package P contains another constraint C' that refers to a class B, which does not exist in P but is imported from package R. If the newly imported package Q also contained a class named B, the semantics of the import operation would make the name B unresolvable (because multiple imports of the same name are not allowed), and hence constraint C' would not work any more. An additional problem with these side-effects is that most tools do not give any warning to the user about them.

3.4 PackageMerge Problems

The PackageMerge operation presents several problems mainly due to its complex semantics, as initially pointed out in [4]. The work in [5] provides a thorough analysis of this operation and its problems, proposing a set of solutions for those that can be resolved, and identifying those that cannot. PackageMerge is not a simple operation, and in fact, the UML documentation discourages its use by casual system modelers. It is said to be intended for expert metamodelers to help them define their metamodels using a modular approach, which allows adding features to existing classes by merging them with others that define the extensions.

Furthermore, to our knowledge there is no general tool support for it. At least we do not know of any UML modeling tool that allows you drawing a ≪merge≫ dependency between two packages and then changes the receiving one according to the result of the operation (as it happens with UML Generalization, despite the fact that the UML manual mentions that these two operations should behave similarly in that way).

4 The Proposal

4.1 A More Flexible Grouping Schema

In order to allow a more flexible grouping schema for modeling elements, we will adopt one of the concepts defined in ODP, the **group**. A group is "a set of objects with a particular relationship that characterizes either the structural relationship among objects, or an expected common behaviour" [6, Part 2, clause 10.1].

Figure 1 shows our proposal of a new grouping schema for UML model elements. The central element is the Group, which is related[1] to a set of GroupableElements that represent the model elements that can be grouped together. The characterizing relation that defines the criteria that the members should fulfil to belong to the group is expressed by an optional UML constraint. All groupable elements (including groups) are named elements (see section 4.2). There is also another constraint (not shown here) that forbids cycles in the membership relation, that is, we do not allow groups to contain themselves or to contain other groups that contain them.

Figure 1 also shows how the Package becomes just an specialization of a Group, in which the members belong to exactly one package (itself). Our proposal allows model

[1] The Membership relationship is represented here by an association, although other UML modellers would have used an aggregation. Given the loose semantics of the UML aggregation, we have preferred to use an association.

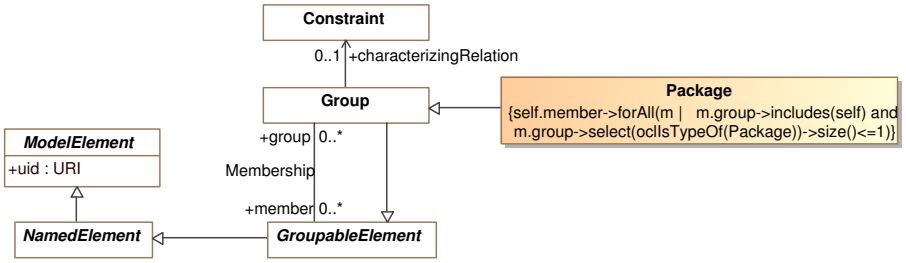

Fig. 1. A new grouping schema for UML model elements

elements to belong to several groups, hence supporting overlapping organizations of model elements. This grouping schema is similar to the classification schema used by Google Gmail, which associates *labels* to mail messages. Such labels act as groups that identify the relations that characterize their member elements. Any Gmail message may have several associated labels, one for each of the groups it belongs to. Selecting a particular label lists all the member messages associated to that label. Similarly, our model elements can be associated to more than one group, avoiding the tyranny of the dominant classification currently required by UML.

4.2 A More Powerful Naming Schema

Our proposed naming schema is based on the ODP Naming standard [8], which provides a *context-relative* naming schema for large, open, federated and heterogeneous systems and environments. In a context-relative naming scheme, multiple naming contexts can apply to entities in different domains. These naming contexts can be related to one another, hence allowing us to refer from one naming context to an entity in another.

To achieve this we need to introduce an intermediate level of indirection between an entity and its name, which is given by a *Naming Action*.

Figure 2 shows the main elements of the proposed naming schema, representing in a class diagram the core part of the ODP naming schema tailored to our specific environment. The elements in that diagram correspond to the concepts defined in [8]:

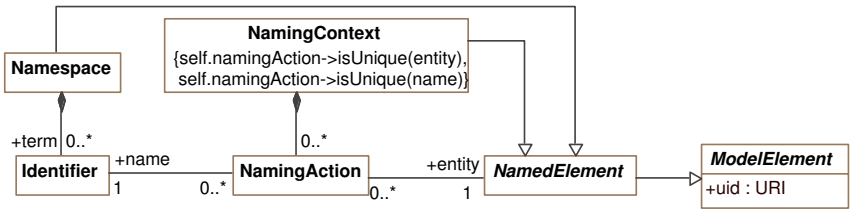

Fig. 2. A new naming schema for UML model elements

- A **name** is is a term which, in a given naming context, refers to an entity. In this paper we will assume that names are represented by Strings, although it would be possible to use other representations such as symbols, images, icons, etc.
- An **identifier** is an unambiguous name, in a given naming context.
- A **namespace** is a set of terms, usable as names, generated by a naming convention.
- A **naming context** establishes a relation between names and entities (in our case, model elements that are subject to naming). Such a relation is composed by a set of pairs "(name,entity)" that defines how the entities are named in this particular context. Such pairs are represented by naming actions. The fact that naming contexts are also named entities enables the relationship between naming contexts.
- A **naming action** is an action that associates a term from a name space with a given entity. All naming actions are relative to a naming context.

Note that for simplicity in this paper we do not allow direct synonyms (entities with several names) nor homonyms (names associated to several entities) within the same naming context, as expressed by the two constraints in the NamingContext class. These constraints could be relaxed to take into account not only the names but also the element types when comparing elements, although in this paper we will use just names.

A very simple and common example of the use of multiple naming contexts can be found in mobile phones. Every person assigns a name to each number in his/her mobile phone's contact list, hence defining a unique naming context. For example, the name I have assigned in my cellular phone to my wife's phone number ("wife") is different from the name that each of my children have assigned to it ("mum" and "mammy", respectively). But we all refer to the same number. Analogously, the same name "ICE" is normally used in all phones to refer to the number to call in case of an emergency, although this number is different for almost everyone. In our terminology, each phone defines a naming context, and each entry in a phone's contact list is nothing but a naming action in that context.

Another example of naming contexts can be found in natural languages. Figures 3 and 4 show two examples of this, using our representation in UML. The first one shows how two model elements are named in the English context. The second figure shows one element being named in six different contexts (only the naming actions are shown, with the references to the related elements shown in their slots).

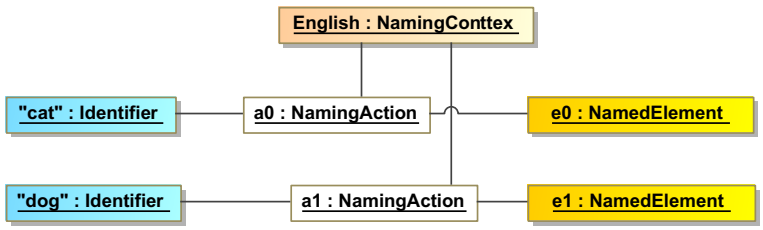

Fig. 3. An example of a naming context with two entities and their associated names

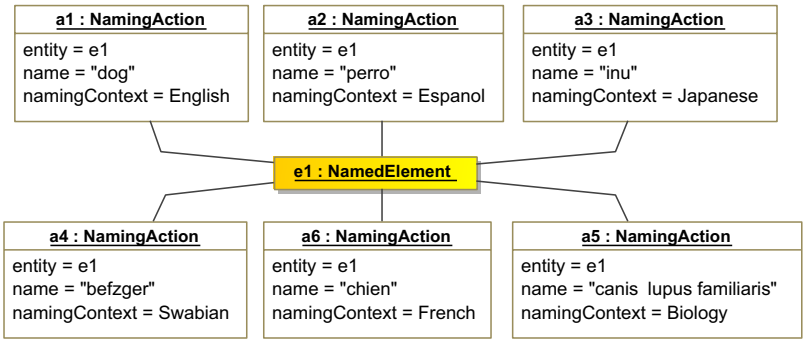

Fig. 4. An example of multiple names for an entity in several naming contexts

The fact that a naming context can be itself named in another context implies an interesting level of indirection by nesting naming contexts. In this way entities that belong to more that one nested domain may be referred to using different names. For example, figure 5 shows how the entity called "dog" in English and "perro" in Spanish, can also be called "Inglés::dog" in the Spanish naming context, because the English context is called "Inglés" in Spanish. Similarly, from the English context a dog can be called both a "dog" and a "Spanish::perro". (We are using "::" as separator for composing names.)

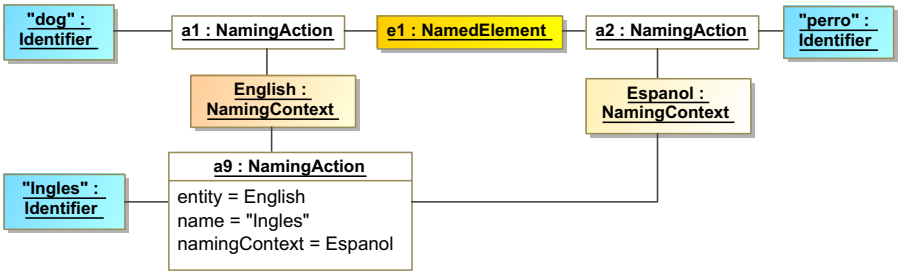

Fig. 5. An example of multiple names for an entity in several naming contexts

4.3 Name Resolution

As defined in ODP, name resolution is the process by which, given a name and an initial naming context, an association between a name and the entity designated by the name can be found. In general this is a difficult process, because it can potentially match multiple elements. In our current proposal naming contexts do not allow synonyms, so the name of every model element is unique in that context. Then we assume that its name will be the only one defined by the appropriate naming action. The following operation resolve() implements name resolution:

```
context Identifier::resolve(C : NamingContext) : NamedElement
   body: self.namingAction->any(namingContext = C).entity
```

Of course, this operation can evaluate to UndefinedValue if the name does not correspond to any model element in the context (for instance, "bfgtrr".resolve(English). This can be checked by operation canResolve():

```
context Identifier::canResolve(C : NamingContext) : Boolean
   body: self.namingAction->exists(namingContext = C)
```

The opposite operation name(), that given a model element and a context returns a name, is not so simple. We could naively think of the following operation:

```
context NamedElement::name(C : NamingContext) : Identifier
   body: self.namingAction->any(namingContext = C).name
```

However, it can also be the case of a model element that is not named in a given context C, but there is another context D in which the element is named (as, say, "x"), and the context D is named "d" in C. Then, the element can be indirectly named in context C as "d::x". Thus, finding the name of an element which is not directly named in a naming context may result in 0, 1 or more names for the element (depending on how many other contexts define a name for it). This is also a complex process because it may involve cycles (if context D names context E and context E names D) which need to be detected to avoid infinite names. Besides, names in general federated schemata can have different incompatible naming conventions. Here we propose a simple approach when looking for names, returning the first one we find if the name can be resolved in the naming context, either directly or indirectly in other referenced naming contexts. For a discussion of the general process for name resolution, the interested reader is referred to the ODP Naming standard [8].

4.4 Putting It All Together

Figure 6 shows the complete picture of our proposal. Once we have individually defined the grouping and naming mechanisms, the question is how to combine them.

We mentioned in section 4.1 that a Group is a NamedElement (see also figure 1), and therefore it can be addressed by its name, being part of the naming schema. Moreover, it is natural to expect that a group provides a naming context for their member elements (although not in an exclusive way, as the UML Package does). Thus we have defined a relationship between Group and NamingContext that establishes how the group assigns names to its members. A Group is related to one NamingContext, the one that provides unambiguous names to its members. The cardinality at the other end of the association means that a NamingContext can provide names for more than one group. That is, several groups can of course share the same naming context for assigning names to their members.

The following constraint forces all members of a group to be named in the naming context associated to the group. If we want a group with elements from different naming contexts, it is just a matter of creating a naming context that imports them (see 5.1).

```
context Group inv AllMembersNamed:
self.namingContext.namingAction.entity->includesAll(self.member)
```

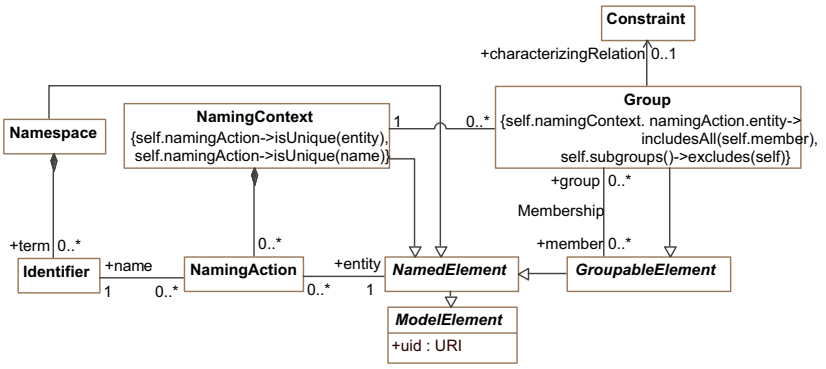

Fig. 6. A summary of the proposed combined naming and grouping schemata

5 Back to the Future

Let us discuss in this section how our proposal can fit with the current UML language and its Package element. In section 4.1 we already mentioned that a package can be considered a special kind of group, whose members belong to exactly one group (itself). We showed this in figure 1, where Package inherits from Group.

Figure 7 provides more details, showing how our proposal can extend the current UML PackageableElement metaclass and its relation with Package. We can see how PackageableElement inherits from GroupableElement, as Package inherits from Group. Similarly, the composition relation between a package and its members is just a subset of the Membership association.

Regarding the provision of a namescope to its members, in the current version of UML, metaclass NamedElement contains an attribute called name (of type String) with the name of that element. In our proposal this attribute has been moved to a property called name of the naming action that assigns a name to an element in a given context, which plays the same role. Still it is obvious to derive the value of the name attribute for a PackageableElement using the relationship between its owner package (which is a Group) and the associated NamingContext.

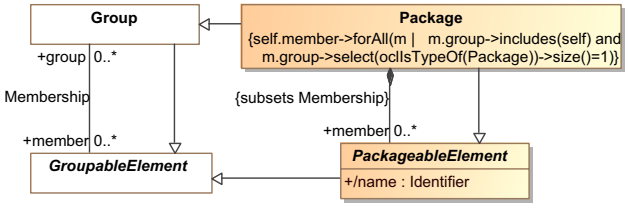

Fig. 7. Extensions to current UML metaclasses

```
context PackageableElement::name : Identifier
  derive: self.name(self.group.namingContext))
```

Furthermore, the following invariant should hold for packages and their members:

```
context Package inv:
  self.member->forAll(m | m.name = m.name(self.namingContext))
```

Note that, in that invariant, the operation name() is always defined for the context associated to the Package, because the AllMembersNamed constraint holds.

Finally, the selection of a concrete syntax for the new Group element is outside the scope of this paper, although any variation of the concrete syntax of the UML Package that showed that overlapping is possible could work well. For simplicity we will use the UML symbol for Package, stereotyped as ≪group≫ to represent groups.

Packages and groups can be used together in a specification. In fact, we think that they provide their best advantages when combined together because each one has its own specific goal. Thus, a very few packages can be responsible for owning the modeling elements, managing their lifecycles and also providing the basic naming contexts for them. Groups become very useful for organizing the model elements according to different categories and/or views in a flexible and structured way.

5.1 Redefining Package Import

The two importing operations in UML are defined by ElementImport and PackageImport relationships. Their goal is to be able to use unqualified names to refer to external members that belong to other packages [1, clause 7.3.39].

The need for such importing operations in UML is due to the strong coupling in UML between names and elements, and between elements and packages. In our proposal we have loosened these relationships, decoupling names from elements, and elements from their groups. Therefore, to use an unqualified name to refer to an entity from a group, in our proposal it is enough to add its name to the naming context associated to the group.[2] The following operation specifies this action. It returns a boolean value indicating whether the operation has succeeded or not.

```
context NamingContext::
          import(n : Identifier, e : NamedElement) : Boolean
  post: if (self.namingAction.name->excludes(n)
            and self.namingAction.entity->excludes(e))
        then result = true and self.namingAction->
               includes (na | na.name = n and na.entity = e)
        else result = ( n.resolve(self) = e )
        endif
```

This operation checks whether the name or the entity already exist in the naming context, but are not related to each other, and if not, a naming action that relates them is added. Otherwise the operation returns false, meaning that the operation has failed.

[2] This operation reflects what normally happens in natural languages when they adopt a term from other languages if they do not have a precise name for a given entity. In English, examples of these names include expresso, siesta, baguette or bonsai, to name a few.

This operation also provides the possibility of importing a complete NamingContext, because a naming context is itself a NamedElement.

Despite it now being an operation on naming contexts, we could also define the corresponding operation import() on groups:

```
context Group::import(n : Identifier, e : NamedElement): Boolean
  body: self.namingContext.import(n,e)
```

Then, if we had a group A that imports the names of the member elements of a group B (via an ≪import≫ relation), the semantics of this operation is now given by the OCL expression A.import(B). Note that this operation has a possible side-effect if several groups share the same naming context: if one group imports an element, it becomes imported to all the groups. However, this is not a problem because of the difference between *importing* one element and *including* it as a member of a group. Importing an element allows referring to it by its name, i.e., it is an operation on naming contexts and not on groups. Including an element (using operations include() or union(), for instance) means adding it to the set of members of the group. Elements can be added by identifier, name or both. We show here the include() operation using identifiers. The other operations are similar. Remember that constraint AllMembersNamed requires included elements to be named in the corresponding naming context.

```
context Group::include ( n : Identifier )
  pre: n.canResolve(self.namingContext)
  post: self.member->includes(n.resolve(self.namingContext))
```

One of the benefits of having a well-defined semantics of all these operations specified in OCL, is that they can now be easily implemented and checked by UML modeling tools. Moreover, the fact that these operations return a boolean value indicating whether a conflict has been found or not during the import operation allow modeling tools to warn users about conflicts—in contrast to current UML PackageImport and ElementImport operations, which do not warn the users about their side-effects in case of conflicts. This is of special importance in large models with many elements, possibly developed by different people in different locations, in which name conflicts can easily pass unnoticed.

The way in which the conflicts are resolved falls outside the scope of this paper, and depends on what we want the import() operation to do in case of name clashes. To illustrate this issue, let us suppose two groups One and Two, each of them with a different naming context, and with four elements that globally refer to six entities $\{e_1, ..., e_6\}$. This is informally shown in figure 8, where dependencies represent graphically the naming contexts of these two groups before the import operation:

One.namingContext $= \{(\mathsf{A}, e_1), (\mathsf{B}, e_2), (\mathsf{C}, e_3), (\mathsf{D}, e_4), ...\}$

Two.namingContext $= \{(\mathsf{A}, e_1), (\mathsf{B}, e_3), (\mathsf{C}, e_5), (\mathsf{E}, e_6), ...\}$

If we want group One to import the names of Two (for calculating the union of both, for example), two conflicts appear for names B and C, each of them with a particular kind of problem. It is clear that after the import operation the resulting naming context of One will contain naming actions $\{(\mathsf{A}, e_1), (\mathsf{D}, e_4), (\mathsf{E}, e_6)\}$. However, the way to resolve the other two names and how to create a new name for the sixth entity depends solely on the user, who should decide what to do with the clashes.

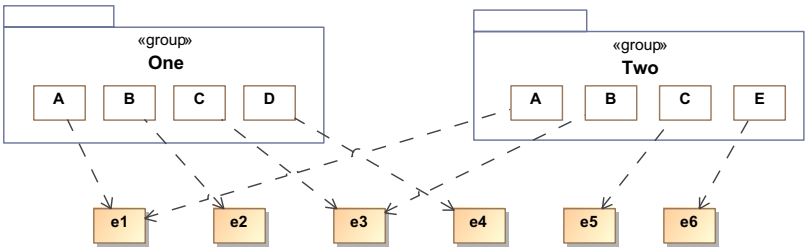

Fig. 8. An example of two overlapping groups

Is it worth noting that in our proposal one could, as an alternative, just name the contexts in each other, and thus avoid the import operation altogether.

5.2 Merging Groups

Decoupling the names from the entities they refer to allows to uncover a semantic distinction between the import and merge operations. While the former deals with the names of the model elements, the latter deals with the elements themselves (as it happened with include and union operations). In fact, several elements of the UML packages to be merged are expected to share names, because they are supposed to represent the same entities but from different perspectives. In theory a package merge is just aimed at the *extension* of the receiving package with the duplicate-free union of the two packages. This should work in a similar way as the Generalization. However, when it comes to the details the situation gets really complex. The handling of the constraints on the model elements and the potential inconsistences between conflicting requirements on the elements to merge complicates the semantics of this operation (see the good discussion presented in [5]). In addition, Generalization works on classes, while PackageMerge works on packages, i.e., sets of model elements. This is a subtle but crucial difference that introduces further problems. Finally, the side-effects of this operation on the receiving package (whose elements are implicitly, but not explicitly modified) complicates the naming operations and the navigation of elements, as mentioned above.

We propose a different approach to group (and thus package) merging, based on the creation of a new group whose members are the result of the merge. These elements can be either previous elements (if they are not modified by the merge—for instance those that belong to just one of the groups) or newly created elements, resulting from the individual merges. On the one hand this approach allows the use of a more powerful merge operation between individual elements, such as the ones proposed for merging DB schema, generic model merging or ontology merging defined by Pottinger and Bernstein in [10], the class merging operation as defined in MetaGME [11], or the epsilon merge operation [12]. On the other hand, this approach allows groups and elements to maintain their properties and avoids undesirable side-effects. The fact that each group can have its own naming context solves the problem of finding names to the new entities created after the merge: they maintain in the new context the names they had before.

Figure 9 shows how the new operation works on the two groups described in figure 8. We can see that the resulting group has 5 elements (and not six as it would happen if instead of merging we had used the union operation), named {A..E}. The elements these names refer to is specified by the resulting naming context:

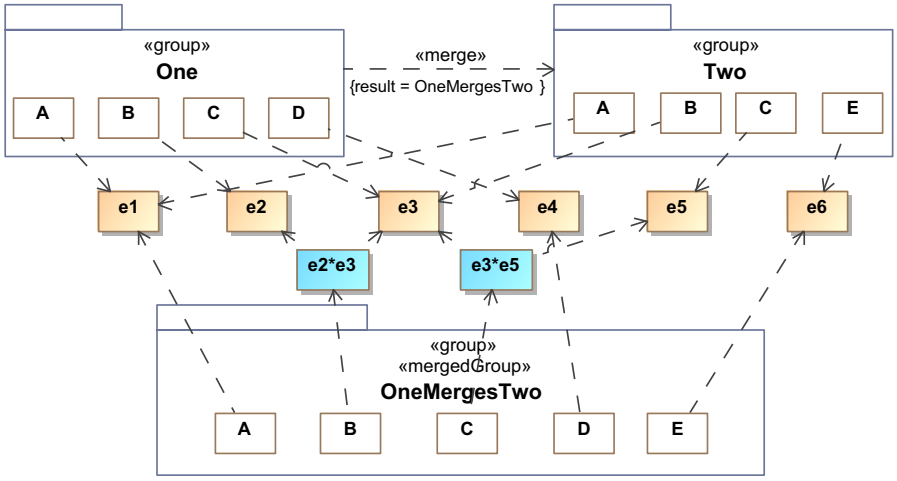

Fig. 9. Merging the two models

$$\text{OneMergesTwo.namingContext} = \{(\text{A}, e_1), (\text{B}, e_2 * e_3), (\text{C}, e_3 * e_5), (\text{D}, e_4), (\text{E}, e_6)\}$$

Name A refers to the same element e_1 in groups One and Two and therefore it remains the same in the resulting group. Names D and E become part of the result without changes because they belong only to one of the two groups to merge. Names B and C are part of the resulting group, but they now refer to the elements obtained by merging e_2 with e_3 and e_3 with e_5, respectively (whose ids we have called e2*e3 and e3*e5). As mentioned before, the way in which these elements are calculated from the original ones is outside the scope of the paper. In general, any merging algorithm can do the job, although the simplest way would be to use one based on multiple inheritance such as the one described in [11].

A new tag definition (result) of stereotype ≪merge≫ on the dependency identifies the group that receives the result of the merge. Such a group should not exist beforehand, and is created to accommodate the merge. The new group is stereotyped ≪mergedGroup≫ precisely to indicate that it is the result of a merge. This stereotype has two tag definitions (named receivingGroup and mergeGroup, not shown in the diagram), that refer to the two groups involved in the merge operation.

This merge operation does not provide full backwards compatibility with the original PackageMerge operation, in order to avoid some of its problems. The new operation does not have any side-effects, and also is flexible w.r.t. the way in which the elements are merged. Furthermore, it is worth noting that modifying the members of a group as a result of a merge operation does not make sense in this new context, because a group does not own its members—it only provides references to them. One could still define

another semantics for this operation that behaved as the previous one in the case of packages (which own their members).

6 Related Work

This proposal is influenced by the naming concepts developed by ISO/IEC for the RM-ODP framework [8], which in turn come from the ANSA project in the nineties [13]. These concepts were defined to cope with the heterogeneity of naming schemata existing in any open distributed system, with possible federated applications, and prove to be very useful for providing a more flexible naming schema to UML—required to enable, for example, federated modeling with UML.

There have always been several currents in the modeling community to improve UML. Probably the most active proposals now come from the OMG's UML revision task force that works on the definition of UML 3; from the group working on fUML [14] for a proper and well-founded subset of the UML 2 metamodel—which provides a shared foundation for higher-level UML modeling concepts; and from the OMG's Architecture Ecosystem SIG [2], whose goal is to manage the alignments between different OMG recommendations (such as UML and SysML, or UML and BPMN), and also aims at achieving federated modeling [15]. The proposal presented here aims at contributing to the work of these groups.

The merging of groups and packages is influenced by our previous work on combining Domain Specific Visual Languages [16]. Model merging can provide a solution to language combination in some cases, and package merge is one of the possible ways to achieve model merge. Regarding the way the individual elements are merged, we allow the use of any of the existing algorithms, such as the ones described in [10,11,12].

7 Conclusions

In this work we have presented a proposal for new UML naming and grouping mechanisms, which are currently provided in UML by the single concept Package. Our proposal addresses some of the limitations that the UML mechanisms have, because UML tightly couples elements with names, and names with packages. Our proposal allows elements to belong to more than one group, and to have more than one name in different contexts; it can accommodate different naming schemata and permit their interconnection while maintaining name unambiguity in each context; and enables the proper definition of grouping and naming operations such as import. One of the major advantages of our proposal is that it tries to respect as much as possible the current version of UML, extending it for backwards compatibility. A second major strength is that it is faithfully based on international standards (by ISO/IEC and ITU-T), making use of mature ideas which are developed with the consensus of major international parties and organizations. Plans for future work include dealing with synonyms and with UML inheritance as part of naming context.

Acknowledgements. The author would like to thank Peter Linington, Martin Gogolla and Pete Rivett for their very useful comments and criticisms on earlier versions of this paper, and to the anonymous reviewers for their helpful suggestions. This work is supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.

References

1. Object Management Group. Unified Modeling Language 2.3 Superstructure Specification (May 2010), OMG doc. formal/2010-05-05
2. Object Management Group. Architecture Ecosystem Special Interest Group (December 2009), http://www.omgwiki.org/architecture-ecosystem/
3. Chimiak-Opoka, J.D., Demuth, B., Silingas, D., Rouquette, N.F.: Requirements analysis for an integrated OCL development environment. In: Proc. of the OCL Workshop at MODELS 2009. ECEASST, vol. 24 (2009)
4. Zito, A., Diskin, Z., Dingel, J.: Package merge in UML 2: Practice vs. Theory? In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 185–199. Springer, Heidelberg (2006)
5. Dingel, J., Diskin, Z., Zito, A.: Understanding and improving UML package merge. Software and System Modeling 7, 443–467 (2008)
6. ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing, ISO/IEC 10746, ITU-T Rec. X.901-X.904 (1997)
7. ISO/IEC. Information technology – Open distributed processing – Use of UML for ODP system specifications, ISO/IEC IS 19793, ITU-T X.906 (2008)
8. ISO/IEC. Information Technology — Open Distributed Processing — Naming Framework, ISO/IEC IS 14771, ITU-T Rec. X.910 (1999)
9. Object Management Group. Object Constraint Language (OCL) Specification. Version 2.2 (February 2010), OMG doc. formal/2010-02-01
10. Bernstein, P.A., Pottinger, R.A.: Merging models based on given correspondences. In: Proc. of VLDB 2003, Berlin, Germany, pp. 862–873 (2003)
11. Emerson, M., Sztipanovits, J.: Techniques for metamodel composition. In: Proc. of the Workshop on Domain Specific Modeling at OOPSLA 2006, pp. 123–139 (2006)
12. Kolovos, D., Rose, L., Paige, R.: The Epsilon book, U. York (2010), http://www.eclipse.org/gmt/epsilon/doc/book/
13. van der Linden, R.: The ANSA naming model. Architecture report APM.1003.01, ANSA (July 1993), http://www.ansa.co.uk/ANSATech/93/Primary/100301.pdf
14. Object Management Group. Semantics of A Foundational Subset For Executable UML Models (fUML). Version 1.0 – Beta 3 (June 2010), OMG doc. ptc/2010-03-14
15. Casanave, C.: Requirements for Federated Modeling. White paper. Model Driven Solutions (January 2011), http://bit.ly/dUGiVf
16. Vallecillo, A.: On the combination of domain specific modeling languages. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 305–320. Springer, Heidelberg (2010)