A Relational Approach to Defining Transformations in a Metamodel

David Akehurst and Stuart Kent

Computing Laboratory, University of Kent, Canterbury, UK
dha@ukc.ac.uk, sjhk@ukc.ac.uk

Abstract. Metamodelling is becoming a standard way of defining languages such as the UML. A language definition distinguishes between concrete syntax, abstract syntax and semantics domain. It is possible to define all three using a metamodelling approach, but it is less clear how to define the transformations between them. This paper proposes an approach which uses metamodelling patterns that capture the essence of mathematical relations. It shows how these patterns can be used to define both the relationship between concrete syntax and abstract syntax, and between abstract syntax and semantics domain, for a fragment of UML. A goal of the approach is to provide a complete specification of a language from which intelligent tools can be generated. The extent to which the approach meets this goal is discussed in the paper.

1 Introduction

In this paper we use the term 'metamodel' to mean the rendering of a language definition as an object model. Metamodelling is perhaps best know in its application to the UML where a subset of UML is used to define (the abstract syntax of) itself [12] – hence the use of 'meta'. Recently, it has been demonstrated that metamodelling can be used to define concrete syntax and semantics, as well as abstract syntax [5][7][1][4][14]. This involves defining a model of concrete syntax and a model of the semantics domain, in addition to a model of abstract syntax, and then modelling, somehow, the transformation relationship between these three components. So far, this has been done in a rather ad-hoc way, using associations that directly connect elements in one component with elements in another, and then writing constraints on one or other side of the association. This paper takes a more systematic approach that has the following two advantages:

(a) It locates all aspects of the relationship in a separate part of the metamodel, thereby achieving a better separation of concerns and avoiding clutter in the metamodel for the components being mapped.

(b) A relationship model need not be biased towards one or other direction in the relationship. This facilitates the construction of bidirectional mapping tools.

With regard to (b), we are particularly interested in automatically generating tools to support modelling with a language from the metamodel of that language. We give two examples, both of which require bidirectional mappings.

The first example is a model editor, where a model is input and viewed through one or more concrete syntaxes (e.g. a textual and diagrammatic syntax), and where the (abstract syntax of the) model can be directly manipulated, e.g. through property

panes in a GUI. For such a tool, it must be possible to change the model using one medium (e.g. property panes) and generate or update views of the model in other mediums (e.g. diagrams). Here, bidirectional mappings between concrete and abstract syntax (AS and CS) are required, so changes in one can easily be propagated to the other.

The second example would be a tool that supports the exploration of a model by allowing one to build examples and counter-examples, that could, for example, be shown to domain experts to validate the model. These examples could be generated from a model, or constructed directly by hand, which requires the mapping from model to example to be precisely specified. There is also the possibility of checking given examples against a model and (partially) generating the model from examples, requiring the mapping to be bidirectional.

Our approach is based on a very simple idea: take the mathematical model of relations, and encode it as an object model. This leads to a particular style or pattern of modelling, which, amongst other benefits, can guide the metamodeller in what they need to consider when defining a mapping. The paper is structured as follows. Section 2 gives the metamodel for the components to be related – concrete syntax, abstract syntax and semantics domain – for a fragment of UML. Section 3 describes the general approach to expressing transformation relationships, by adopting a modelling style inspired by mathematical relations. Section 4 shows the application of the approach to defining the relationships between concrete and abstract syntax, and between abstract syntax and semantics domain. Finally, Section 5 points to ongoing and future work, including the use of a package template mechanism, which can be automated, to encode the modelling styles described here, the development of specialised notation for describing such mappings, the generation of mapping tools from the definitions and the application of the techniques to modelling relationships or transformations in general.

2 Metamodels for AS, CS and Semantics Domain

Metamodels in this paper are expressed using a subset of UML comprising class diagrams and OCL for writing constraints. This subset corresponds closely with the language used by MOF [11] to define metamodels, and is similar to that implemented (and formalised) by the USE tool [15]. In this section we define the metamodels for three components of the language fragment, between which we will then model relationships. The language fragment contains packages, classes and associations, which map down onto a semantics domain of snapshots, objects and links. A simple model of generalisation between classes is included.

2.1 Abstract Syntax

The abstract syntax is given by Fig. 1. A package may contain classes, associations and other packages. An association contains two association ends (we consider binary associations only). The diagram would be accompanied by a series of OCL constraints which we'll summarise here for brevity:

- A package may not be contained in itself (not a member of allNested).
- upper >= lower for association ends.
- A class can not be a direct or indirect parent of itself (in allParent).

– The names of association ends of which the same class is the source must be unique.
– The names of classes in a package must be unique.
– The names of associations in a package must be unique.
– The names of packages in a package must be unique.
– For an association, associationEnd.otherEnd results in the same set.
– For an associationEnd, source = otherEnd.target and target = otherEnd.source.
– For an association, the package that minimally contains the source and target of the association ends is the package that contains the association.

The last two constraints avoid situations, for example, where a package A contains a package B which contains classes X and Y, and also contains an association between X and Y. In these situations it would make more sense for the association to be contained in B rather than A.

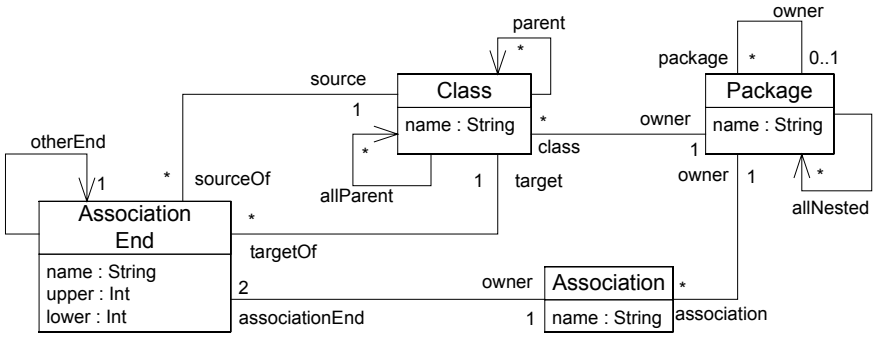

Fig. 1 Abstract Syntax

2.2 Semantics Domain

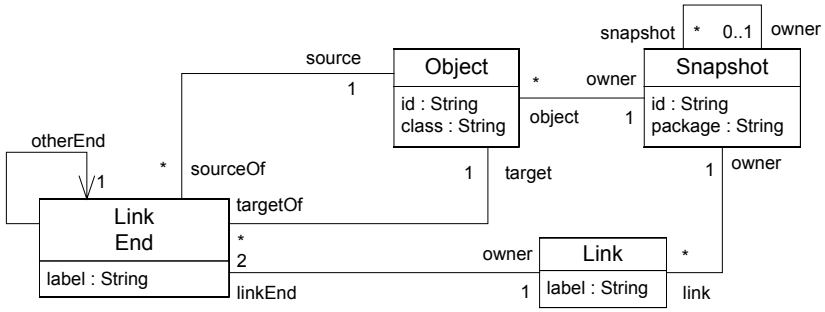

Fig. 2 Semantics Domain

The semantics domain is given by Fig. 2. Instances of packages are snapshots (configurations of objects and links) which contain objects, links and other snapshots. A link contains two linked ends. There are fewer constraints this time:

– A snapshot may not be contained in itself, directly or indirectly

– The snapshot which minimally contains a link must be the snapshot itself.
– Id's of objects in a snapshot are unique; similarly for snapshots in a snapshot.
– For a link, linkEnd.otherEnd results in the same set.

This time there are no constraints on labels of links or link ends, because it is quite possible, indeed desirable, for there to be more than one link end with the same label of which the same object is the source, and more than one link with the same label in the same snapshot. This is because labels here are used to identify which association end / association the link end / link is an instance of.

2.3 Concrete Syntax

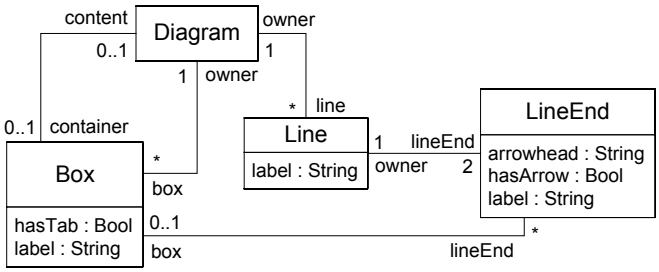

Fig. 3 Concrete Syntax

The metamodel for the concrete syntax is given by Fig. 3. This loses absolute positioning information, but retains relative positioning information: whether or not a line is connected to a box, by virtue of one of its ends being connected to a box, and whether or not a box/line is contained in another box, by virtue of it being part of a diagram that represents the content of a box. Again, we summarise the additional constraints required:

– Lines are contained in the diagram that minimally contains the two ends.
– A box must not contain itself.

3 Patterns for Modelling Transformations

The main inspiration for the technique described here is mathematical relations. The technique is quite simple: adopt a pattern which models a transformation relationship as a (binary) relation (or collection of relations), and encode this as an object model. With this in mind, pairs are modelled as objects, and relations as objects that are associated with a set of pairs. Pairs in a relation may themselves be associated with other relations, and so on recursively. This allows relationships between structures with many levels to expressed, as required in language definition. Each of these aspects is now explained in detail.

3.1 Pairs

Pairs are encoded as objects which are associated with an object from the domain and range, respectively. If domain objects come from the class X and range objects from the class Y, one can define a class XY representing pairs of X's and Y's as given by

Fig. 4. Directed associations ensure that X and Y require no knowledge that they are being "paired up".[1]

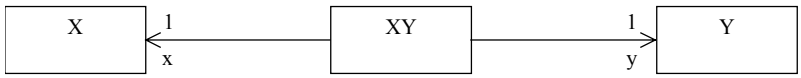

Fig. 4 Pairs

When applying this pattern one substitutes classes for X and Y, and relabels XY and its associations as expected. If domain and range objects come from the same class (say X), then xDomain and xRange can be used as labels on the association ends, instead of x and y. Or some other appropriate naming scheme can be used instead.

3.2 Relations

A (binary) relation is then an object associated with a set of pairs. A relation has a domain and range. All domain objects associated with the pairs that constitute the relation must be selected from the domain, and all range objects from the range. This leads to the class diagram in Fig. 5.

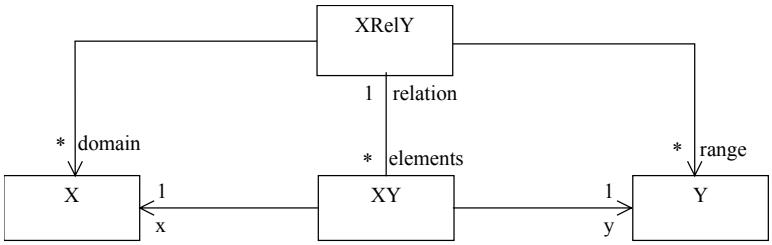

Fig. 5 Relations

Again, associations are only navigable from XrelY so that X and Y remain untouched by the definition of the relation.

Contrary to what might be expected at first sight, the domain and range are not the classes (e.g. X and Y) from which domain and range objects are selected, they are subsets of these classes. This is an important insight whose full explanation is deferred to Section 3.3, which considers how to model transformations between nested structures. The implication is that an association between X and Y is not sufficient for modelling a relation between X's and Y's because it requires the domain and range to be the classes X and Y.

Fig. 5 must be accompanied by an invariant which ensures that the elements of pairs in the relation are selected from the domain and range.

```
context XrelY inv:
  domain->includesAll(elements.x->asSet) and
  range->includesAll(elements.y->asSet)
```

And a further constraint is required to ensure that no two pairs in the relation refer to exactly the same domain and range elements. This avoids redundancy, and simplifies the definition of queries in the sequel.

---

[1] This may be important in a situation where one wishes to define a mapping between models that are already populated in some repository.

```
context XrelY inv:
  elements->forAll( e, f |
    (e.x = f.x and e.y = f.y) implies e = f )
```

It is also useful to define a number of auxiliary query operations (functions) on the class XrelY:

(i) The image is the set of elements from the range actually mapped onto, under the relation.

```
context XrelY::image() : Set(Y)
  post: result = elements.y->asSet
```

(ii) The inverse image is the set of elements from the domain actually mapped from, under the relation.

```
context XrelY::inverse_image() : Set(X)
  post: result = elements.x->asSet
```

(iii) The relation is *onto*, if the image is the range.

```
context XrelY::is_onto() : Boolean
  post: result = ( image = range )
```

(iv) The relation is *total*, if the inverse image is the domain.

```
context XrelY::is_total() : Boolean
  post: result = ( inverse_image = domain )
```

(v) The relation is *functional*, if and only if an element of the domain maps to at most one element in the range.

```
context XrelY::is_functional() : Boolean
  post: result = elements->forAll( p,q |
        p.x = q.x implies p = q )
```

(vi) The relation is *inverse functional*, if and only if an element of the range is mapped to from at most one element in the domain.

```
context XrelY::is_inverse_functional() : Boolean
  post: result = elements->forAll( p,q |
        p.y = q.y implies p = q )
```

(vii) The relation is an injection if it is both functional and inverse functional.

```
context XrelY::is_injection() : Boolean
  post: result = is_functional and is_inverse_functional
```

(viii) The relation is a bijection if it is both an injection, onto and total.

```
context XrelY::is_bijection() : Boolean
  post: result = is_injection and is_onto and is_total
```

(ix) Looking up an element from the domain returns the pairs in the relation which mention that element.

```
context XrelY::domainLookup(X:x) : Set(XY)
  post: result = elements->select( p | p.x = x)
```

(x) Looking up an element from the range returns the pairs in the relation which mention that element.

```
context XrelY::rangeLookup(Y:y) : Set(XY)
  post: result = elements->select( p | p.y = y)
```

When applying this pattern, one substitutes for classes X and Y, relabelling things as appropriate, remembering that variables etc. used in the invariants and definitions of

the queries on XrelY will also need to be relabelled. These patterns could be captured as *package templates*, as originally described in the Catalysis method [8], which would effectively systemise and automate the substitution process. Templates have been already been used to encode patterns like this for metamodelling [1][5].

When applying the pattern, it is usually necessary to add additional constraints stipulating the specific properties of the relation under consideration. These constraints fall into three categories:

- Definition of the properties of the relation – bijective, functional, total, etc.
- Definition of the domain and range.
- Relation specific constraints, with respect to the elements forming the relation content.

The first of these is the simplest to define; it is of the form:

```
context Rel inv: f()
context MyClass inv: myRel.f()
```

where 'f' is a combination of one or more of the queries – injection, bijection, total, etc. For example, one may define a class ArelB according to the pattern described here, than add the additional constraint:

```
context ArelB inv: is_bijection()
```

which will ensure that all relations of the class will be bijections. If it is the case that only some relations of this class should be bijections, then one should apply the constraint directly and only to the object representing that relation. So, for example, if myRel:ArelB is declared in a class MyClass, one might write:

```
context MyClass inv: myRel.is_bijection()
```

The second kind of constraint defines the domain and range of the relation. The constraints must be formed to involve the 'domain' and 'range' navigable features from the relation and are generally of the form:

```
domain = <set expression>
range = <set expression>
```

where 'set expression' is an OCL expression resulting in a collection of objects of the appropriate class. These definitions will nearly always appear as part of the definition of a class which uses a relation. For example, if `MyClass` also has declared `myAs:Set(A)` and `myBs:Set(B)`, one could write:

```
context MyClass inv:
  myRel.domain = myAs and myRel.range = myBs
```

to define the domain and range for `myRel`.

The final kind of constraint is often the most complex. It is often of the form:

```
elements->forAll( e | <expression> )
```

where 'expression' puts constraints on the elements that form the relation content. The detail of the expression is dependent on the overall application and purpose of the specified transformation. Examples of these constraints can be found in section 4.

3.3 Relating Structures

For most practical modelling, it is necessary to combine different relations. The core pattern here is the idea that an element of a relation can itself be associated with other

relations. A typical example of this is the specification of a relationship between two corresponding containers, as illustrated in Fig. 6.

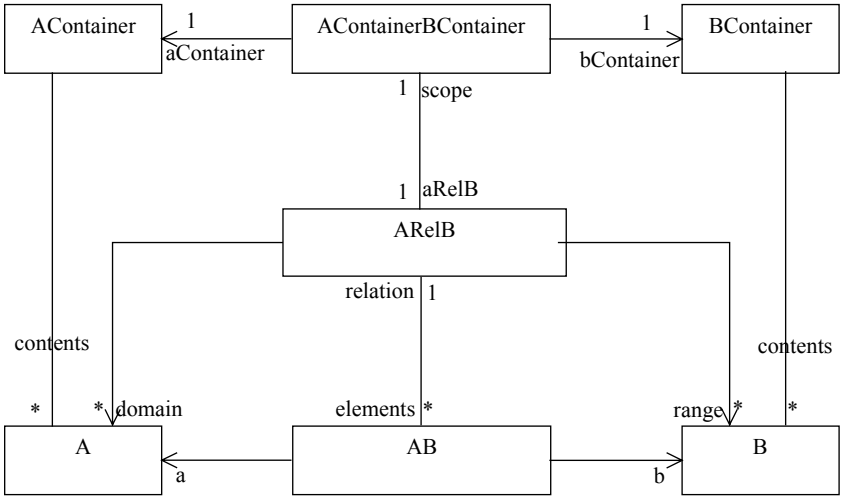

Fig. 6 Relating Structures

This shows the definition of a relationship between a container of A objects and a container of B objects. Given a specific pair of an AContainer and a BContainer, it is necessary to state how the contents of the AContainer map into the contents of the BContainer. This is modelled as a relation between A's and B's. Thus the class AContainerBContainer is associated with the class ARelB. Additional constraints can be added to determine the properties of the relationship between the contents of the containers.

First, the domain of the relation should be the contents of the AContainer and the range should be the contents of the BContainer, as follows:

```
context AContainerBContainer inv:
  aRelB.domain = aContainer.contents and
  aRelB.range = bContainer.contents
```

This illustrates why the domain and range of a relation can *not* be the classes from which the domain and range elements are selected, respectively. If they were we would lose the opportunity to distinguish between the case when the mapping of contents of an AContainer or BContainer covers all the contents or only some of them. As it stands which can choose to force the mapping to cover all the contents of the AContainer by making it *total*, and to cover all the contents of the BContainer by making it *onto*:

```
context AContainerBContainer inv:
  aRelB.total() and aRelB.onto()
```

If the domain and range were set to be the classes A and B, respectively, then the above constraint would only be desirable if the aContainer.contents was the set of all instances of the class A; similarly for bContainer.contents. This will rarely be the case. If we did not model the relation as a class, but instead chose to represent it as an

association between classes A and B, it would be impossible to express the behaviour captured by the invariant above. It would be possible to use an association with a corresponding association class to replace the class AB, however that would begin to interfere with classes A and B, assuming an implementation of a bidirectional association which requires each role end to require a feature (attribute or query) to be added to the class at the opposite end.

Additional constraints may be added to AContainerBContainer to impose further properties on the mapping between contents. For example, to require the relation to be functional, so that every element in aContainer.contents maps to at most one element in bContainer.contents, the following constraint can be used:

```
context AContainerBContainer inv:
  aRelB.isFunctional()
```

The pattern may be repeated to map nested structures, as illustrated by Fig. 7.

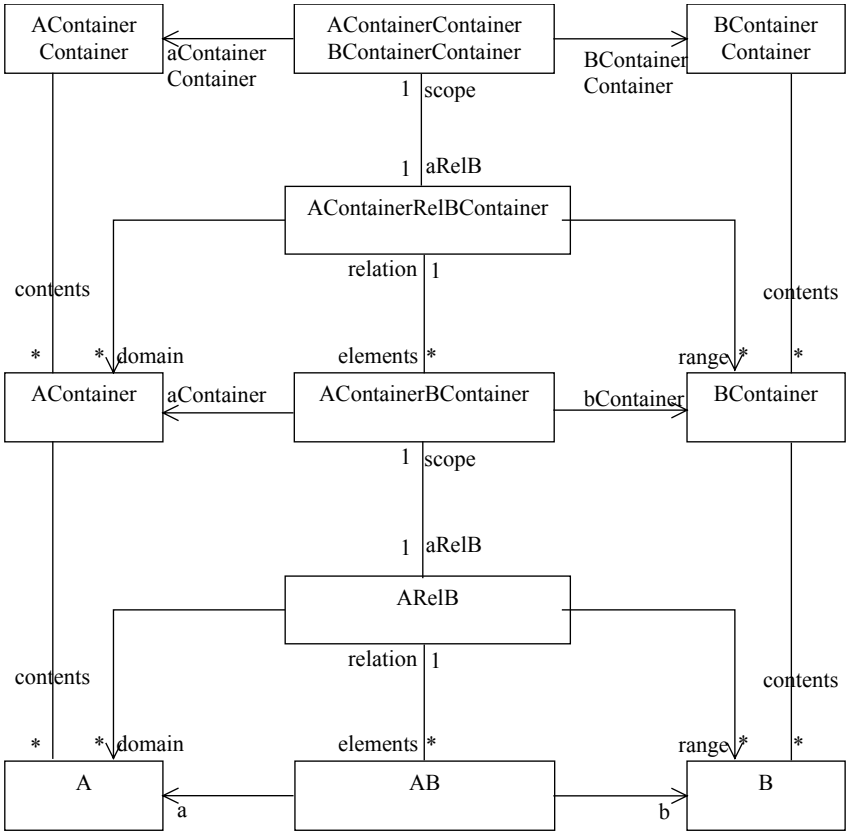

Fig. 7 Transformation between Nested Structures

3.4 Separation of Concerns

A mapping can be made easier to understand and managed in a modelling tool, for example, by wrapping the two sides of the relationship and the relationship itself in separate packages. An example of using packages in this way is provided by Fig. 8.

4 Application of the Patterns

4.1 Abstract Syntax to Semantics Domain

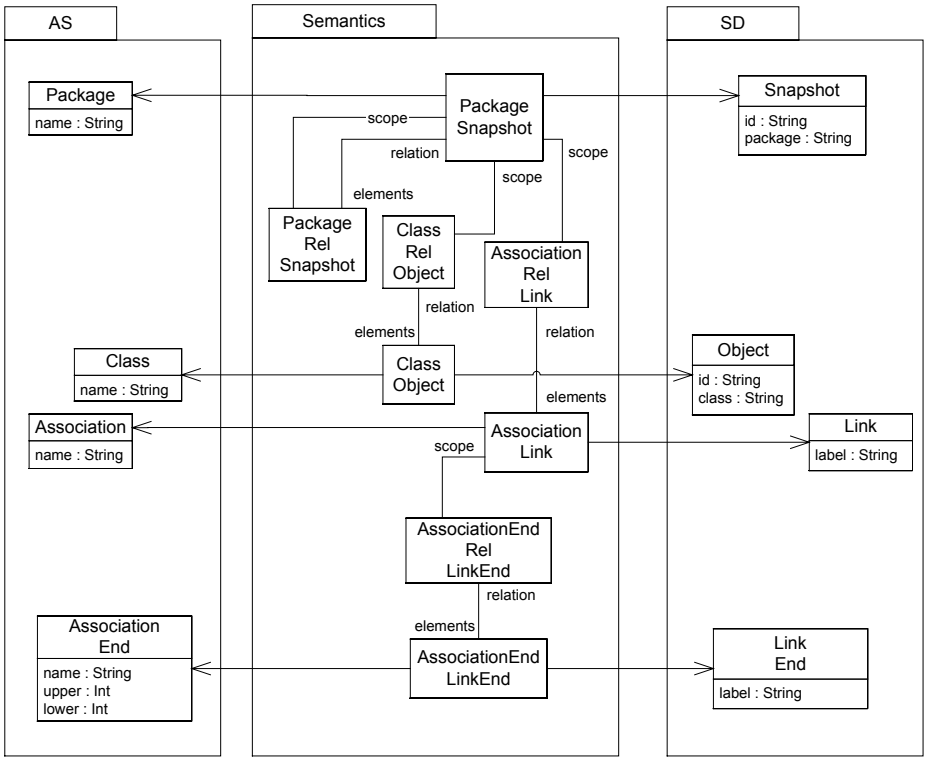

Fig. 8 Semantics Relation

Fig. 8 shows the basic structure of the mapping. We have omitted the details of the abstract syntax (AS) and semantics domain (SD) packages as they are provided in Section 2. We have also omitted cardinalities on associations, where they match the patterns described in Section 3 (which is all of them), and labels on association ends where the label matches the name of the target class. We have omitted domain and range associations (see Section 3 for details). All queries introduced in Section 3 are assumed to be replicated according to the application of the pattern. So, for example, on the class ClassRelObject we have, amongst other queries:

```
context ClassRelObject::image() : Set(Object)
  post: result = elements.object->asSet
```

What remains, is to provide the various constraints according to the guidelines set out in Section 3. First, we define the domains and ranges of the various relations.

```
context PackageSnapshot inv:
  packageRelSnapshot.domain = package.package and
  packageRelSnapshot.range = snapshot.snapshot and
  classRelObject.domain = package.class and
  classRelObject.range = snapshot.object and
  associationRelLink.domain = package.association and
  associationRelLink.range = snapshot.link

context AssociationLink inv:
  associationEndRelLinkEnd.domain = association.associationEnd
  and associationEndRelLinkEnd.range = link.linkEnd
```

Next, come the properties of the relations:

```
context PackageSnapshot inv:
  classRelObject.is_onto() and
  classRelObject.is_inverse_functional() and
  associationRelLink.is_onto() and
  associationRelLink.is_inverse_functional() and
  packageRelSnapshot.is_injection() and
  packageRelSnapshot.is_onto()

context AssociationLink inv:
  associationEndRelLinkEnd.is_bijection()
```

The reasoning behind these decisions runs as follows. All objects must be related to exactly one class (inverse_functional and onto), but some classes may not be represented by objects in a snapshot, and there may be many objects per class (so not total and not functional). Similarly for links of associations. On the other hand, a snapshot of a package which contains other packages may contain snapshots for the latter, but only one of each (functional). A contained snapshot must always be related to exactly one contained package (inverse_functional and onto). This reflects the observation that, at least for in this language, contained packages just allow the namespace of a package to be further partitioned, and this should be replicated in snapshots. Although there may be many links per association, each link must have exactly two link ends which correspond directly with the association ends of the association associated with the link. Thus associationEndRelLinkEnd is a bijection.

Finally, we deal with the constraints specific to this metamodel. First some simple constraints which ensure names tie up correctly:

```
context PackageSnapshot inv:
  package.name = snapshot.package

context ClassObject inv:
  class.name = object.class

context AssociationLink inv:
  association.name = link.label

context AssociationEndLinkEnd inv:
  associationEnd.name = linkEnd.label and
```

Second, a constraint is required to ensure that cardinality of associations is preserved in snapshots.

```
context PackageSnapshot inv:
  package.association.associationEnd->forAll(ae |
  snapshot.allObject()->select(o |
  classRelObject.rangeLookup(o).class = {ae.source})
  ->forAll(o | let n = o.sourceOf->select(le |
   associationRelLink.domainLookup(ae.owner).linkEnd
   ->includes(le))->size in n <= ae.upper and n >= ae.lower))
```

The constraint works by taking each association end ae accessible to the package, then taking each object o accessible to the snapshot of the class at the source of ae, counting all the link ends of ae sourced on o, and checking that this falls within the upper and lower bounds declared in ae. Note the use of range and domain lookups to find pairs in a relation. This allows one to find the elements related in the domain (range) to the range (domain) element in question.

Third, a constraint is required to ensure that a linkEnd is targeted on an object of a class conformant with the class at the target of the corresponding associationEnd:

```
context PackageSnapshot inv:
  associationRelLink.elements->forAll(el |
   el.link.linkEnd->forAll(le | let classOfObject =
    classRelObject.rangeLookup(le.target).class in
    classOfObject->union(classOfObject.allParents)
     ->includes(el.association.associationEndRelLinkEnd
      ->rangeLookup(le).associationEnd.target))
```

4.2 Concrete to Abstract Syntax

Fig. 9 shows the basic structure of the mapping. The diagram is subject to the same conventions and omissions as Fig. 8. Again, we provide the various constraints according to the guidelines set out in Section 3. First, we define the domains and ranges of the various relations.

```
context BoxPackage inv:
  boxRelPackage.domain = box.content.box->select(hasTab) and
  boxRelPackage.range = package.package and
  boxRelClass.domain = box.content.box->select(not hasTab) and
  boxRelClass.range = package.class and
  boxRelPackage.domain->union(boxRelClass.domain)=
      box.content.box and
  lineRelAssociation.domain = box.content.line and
  lineRelAssociation.range = package.association

context LineAssociation inv:
  LineEndRelAssociationEnd.domain = line.lineEnd and
  LineEndRelAssociationEnd.range = association.associationEnd
```

That is, classes are the boxes in the contained diagram that do not have tabs and packages are the ones that do (and these are the only boxes allowed); associations are the lines in the diagram.

Next, come the properties of the relations. At this point we can determine the relationship between the symbols in the diagram and the defined model; for instance, do we allow partial diagrams, and/or model elements to be represented multiple times in a diagram. The following constraints define complete diagrams, with one element mapping to one symbol:

```
context LineAssociation inv:
  LineEndRelAssociationEnd.is_bijection()
```

```
context BoxPackage inv:
  boxRelPackage.is_bijection() and
  boxRelClass.is_bijection() and
  lineRelAssociation.is_bijection() and
```

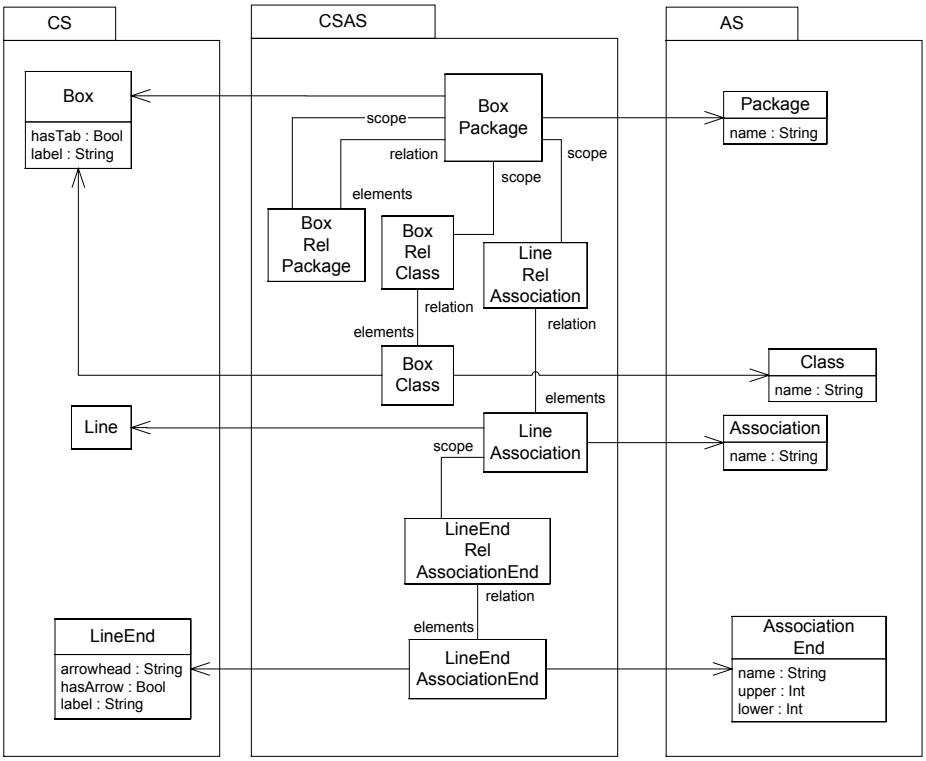

Fig. 9 CS-AS transformation

An alternative, allowing diagrams to be partial views, would be to weaken these properties, by making them total injections, but not onto. To enable elements to be represented multiple times on a diagram, the relations no longer be inverse functional. Associations always have two ends and so must Lines, hence the relationship between LineEnds and AssociationEnds is always a bijection.

Finally, we deal with the constraints specific to this metamodel. First some simple constraints that ensure element names correspond to labels in boxes. The first of these also ensures that the diagram owning the box representing a top level package only contains that box and is not contained in any other box.

```
context BoxPackage inv:
  box.label=package.name and package.owner=null implies
    (box.diagram.box={self} and box.diagram.line->isEmpty
    and box.diagram.container=null)

context BoxClass inv:
  box.label = class.name

context LineAssociation inv:
  line.label = association.name
```

```
context LineEndAssociationEnd inv:
  lineEnd.label = associationEnd.name
```

Second, we must ensure that the boxes at the ends of the lines are mapped to the classes at the ends of the association represented by the line:

```
context BoxPackage inv:
  lineRelAssociation.elements->forAll(el |
   el.line.lineEnd->forAll(le |
     boxRelClass.domainLookup(el.association.associationEnd
       ->one(ae | ae.label = le.label).target).box = le.box)))
```

The constraint works by running through all line-association relationships, and for each one checking that each line-end of the line in the relationship connects to the box that corresponds to the class at the target of the association end that corresponds to that line-end.

5 Conclusions

We have introduced a novel approach to defining transformation relationships between different components of a language definition rendered as a metamodel. This uses a particular style or pattern of modelling that takes its inspiration from mathematical relations. We have demonstrated the approach through the definition of a small, UML-like language. There are a number of possible developments to explore.

- Application of the approach to a more sophisticated language. Although we expect the approach will scale up, we are conscious that other patterns of combining relations will probably be required (e.g. relation composition), and are nervous about the size of the metamodel that might be required.
- Encoding the patterns as package templates [5][6]. This will provide one approach to taming the complexity of large definitions, not least by allowing whole chunks of metamodel to be generated simply through the substitution of template parameters (templates just provide a formal encoding of the systematic rules we have been using when applying the patterns of section 3).
- Development of specialised notation for describing such relationships. This would be an alternative to using templates. It is recognised that metamodelling is missing language constructs for describing relationships between models, including metamodels and their component parts. For example, an OMG RFP to add such a facility to the MOF is soon to be issued. An approach which provides some specialised syntax for the modelling patterns presented here would be a candidate solution.
- Application of the techniques to the OMG's Model Driven Architecture (MDA) [13]. MDA requires transformations between different modelling languages and different models in the same language. It would be a good test of the approach to see how amenable it is to defining such relationships.
- Automated generation / configuration of tools. The idea here is to use a metamodel definition as direct input to a (meta)tool that can then execute and / or monitor the relationship in one or other direction, or generate a tool that can do this. A prototype (meta)tool that generates relationship management tools is described in [2]. This was used to build the diagrammatic language editor for an automaton model checker [3]. This was based on a less flexible method for

modelling relationships [2]. Work is in progress to update the metatool based on the approach described in this paper.

– Alignment with other approaches to language definition. We are particularly keen to explore the relationship between metamodelling and graph grammars [9][10]. In particular, triple graph grammars [16] provide the basis of a rule based approach to expressing mappings that would complement well the metamodelling approach described here.

References

[1] 2U Submitters. Submission to UML 2.0 Infrastructure RFP, available from www.2uworks.org.

[2] Akehurst D. H. Model Translation: A UML-based specification technique and active implementation technique. PhD Thesis, University of Kent, UK. December 2000.

[3] Akehurst D., Bowman H., Bryans J. and Derrick J. A Manual for a ModelChecker for Stochastic Automata. Technical Report 9-00, Computing Laboratory, University of Kent, December 2000.

[4] Alvarez J.M., Clark A., Evans A. and Sammut P. An action semantics for MML. In C. Kobryn and M. Gogolla, editors, Proceedings of The Fourth International Conference on the Unified Modeling Language (UML'2001), LNCS. Springer, 2000.

[5] Clark A., Evans A. and Kent S. Engineering modelling languages: A precise meta-modelling approach. In Proceedings of ETAPS 02 FASE Conference, LNCS. Springer, April 2002.

[6] Clark A., Evans A. and Kent S. Package Extension. Submitted to UML'02, March 2002.

[7] Clark A., Evans A., Kent S., Brodsky S., and Cook S. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Available from www.puml.org, September 2000.

[8] D'Souza D. and Wills A. Objects, Components and Frameworks With UML: The Catalysis Approach. Addison-Wesley, 1998.

[9] Ehrig H., Engels G., Kreowski H-J., and Rozenberg G., editors. Handbook Of Graph Grammars And Computing By Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific, October 1999.

[10] Fischer T., Niere J., Torunski L. and Zündorf A. Story Diagrams: A new Graph Transformation Language based on UML and Java in 6th Int. Workshop on Theory and Applications of Graph Transformation, TAGT'98 Selected Papers (Ehrig, Engels, Kreowski, Rozenberg Eds.) Springer LNCS 1764 (2000).

[11] Object Management Group. The Meta Object Facility (MOF) Version 1.3.1. OMG document number formal/2001-11-02.

[12] Object Management Group. The Unified Modeling Language Version 1.4. OMG document number formal/01-09-67.

[13] OMG Architecture Board ORMSC. Model driven architecture (MDA). OMG document number ormsc/2001-07-01, available from www.omg.org, July 2001.

[14] Reggio G. and Astesiano E. A proposal of a dynamic core for UML metamodelling with MML. Technical Report DISI-TR-01-17, DISI, Universit di Genova, Italy, 2001.

[15] Richters M. and Gogolla M. Validating UML models and OCL constraints. In A. Evans, S. Kent and B. Selic, editors, The Third International Conference on the Unified Modeling Language (UML'2000), York, UK, October 2-6. 2000, Proceedings, LNCS. Springer, 2000.

[16] Schürr A. Specification of Graph Translators with Triple Graph Grammars, in Tinhofer G. (ed.) Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, LNCS 903, Springer Verlag, 151-163, June 1994.