RZ 3574 (# 99584) 02/14/05
Computer Science 21 pages

Research Report

A Layered Approach to Defining a Transformation Language - Informal Description and Validation by Case Study

Ksenia Ryndina and Jochen M. Küster

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{ryn, jku}@zurich.ibm.com

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

A Layered Approach to Defining a Transformation Language - Informal Description and Validation by Case Study

Research Report RZ 3574

Ksenia Ryndina and Jochen M. Küster

Computer Science Department
IBM Zurich Research Laboratory
CR-8803 Rüschlikon, Switzerland
{ryn,jku}@zurich.ibm.com

1 Introduction

Model transformation is the process of changing a given *source* model to produce a *target* model, according to a pre-defined set of rules. Usually transformations modify model structure, notation or level of abstraction, while preserving the underlying semantics. Certain transformations create a completely new target model, while others only update the source model.

The most commonly known transformation examples include generation of code from software design models and code refactoring. Code generation and refactoring features are offered by many software development tools, and established techniques for implementing such transformations are available to tool developers. In recent years however, all software engineering activities have started to become more model-oriented and a great need for transformation between various modelling notations has emerged. The driving force behind these software engineering trends is the *Model Driven Architecture (MDA)* [mda02], which is based on the idea of automated transformation between Platform Independent Models (PIM) and Platform Specific Models (PSM). As a result of the increased demand for more diverse model transformations in the industry, there arises a need for languages, methods and tools that can assist in development of such transformations.

Several efforts have already been made to address the problem of model transformation development. Existing work in this area includes the *Query/Views/Transformation (QVT)* specification [Obj04] by the Object Management Group (OMG) and the *Graph Rewriting and Transformation (GReAT)* language by Agrawal *et al* [AKL03,Agr04].

This report is based on our case study of developing a significant model transformation for the IBM WebSphere Business Integration Modeler. Our main objective during the case study was to identify the requirements for a generic model transformation language and explore the alternative ways in which these can be satisfied.

The transformation that we explored during the case study was part of the cycle-removal procedure for process models that conform to the *Business Operations Metamodel (BOM)* [FS04]. The source and target models for this transformation were in the same notation, and each rule in the transformation was an update of the source model.

Our initial requirements for the transformation language were usability and expressiveness. With this in mind, we decided to adopt a visual, metamodel-based approach to describing transformation rules. Exploring ideas from the existing work in the area, at the end of the case study we formulated a generic transformation language that incorporates some elements from the work of Milicev [Mil02b,Mil02a] and Agrawal *et al* [AKL03,Agr04]. In the remainder of this report, we present this transformation language using examples from the case study. Note that the language is targeted at expressing update transformations at the moment, but in the future we intend to generalise its application to transformations that create completely new target models.

The next section gives an overview of our transformation language by introducing its most essential concepts and their relation to each other. This is followed by a detailed description of the language. Further, we provide an overview of our experience with the case study as validation of the presented transformation language. Several suggestions for future work are given at the end to conclude the report.

2 Language Overview

A transformation language needs to provide a means for rule expression, as well as allow one to describe the order in which rules should be applied to a source model. We adopt a multi-tiered approach for separation of such different concerns of a transformation language. An overview of the language in terms of its tiers is given in Figure 1. The four shaded blocks stacked on top of each other represent the tiers that our transformation language comprises. The main concepts related to each tier are shown as dashed blocks in the diagram.

The highest level of our language is concerned with *transformation units* [Kus00] [KHE03], where the control flow or the order in which transformation rules are applied to a model is determined. It is often the case that at a specific point in time, the same transformation rule can be applied at different locations within a source model. One of the control flow issues is choosing between such alternative locations. Another related control flow issue is establishing when a particular transformation process should terminate.

The tier below the control flow deals with individual transformation rules. In this context, a "rule" is a precise description of the desired transformation in terms of source and target model elements. Each transformation rule consists of a *left hand side (LHS)* and a *right hand side (RHS)*. The LHS and RHS of a rule describe the model under consideration before and after the transformation rule is applied. The steps or operations required to perform the actual transformation are not stated, which gives our approach to rule expression a declarative nature.

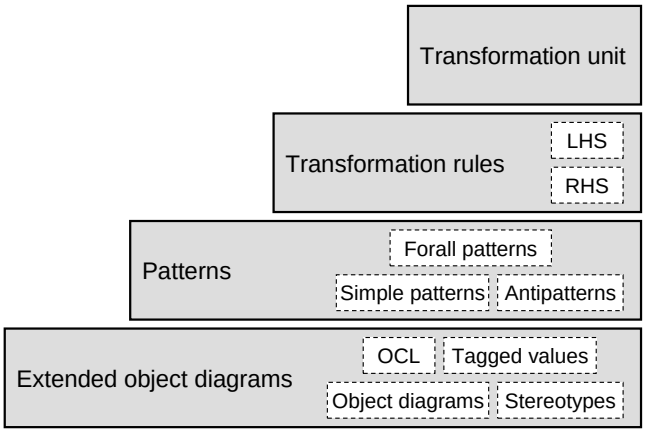

Fig. 1. Transformation Language Overview

Essentially, the LHS defines the pre-conditions for a transformation rule and the RHS defines its post-conditions. Implementation of transformations captured in this manner requires an algorithm for updating a given source model to produce a target model that satisfies the post-conditions expressed in the rule's RHS.

Declarative description of models is achieved by using *patterns*, which are handled by the second-lowest tier in Figure 1. In this context, a "pattern" is defined as a set of modelling elements connected to each other with valid associations from the underlying metamodel. If an occurrence of a pattern is found in a model, then it is said that the model contains a *match* for that pattern. Three types of pattern constructs are shown in Figure 1: *simple patterns*, *antipatterns* and *forall patterns*. Simple patterns can be used to check that certain elements and relations occur within a model. On the other hand, antipatterns check that a model "does not" contain certain elements and relations. Finally, forall patterns allow one to determine whether a constraint holds on a collection of elements of the same type found in a model.

Each pattern is represented graphically as an extended object diagram, using the UML-related constructs shown in the lowest tier in Figure 1. Objects in pattern diagrams instantiate classes from the underlying metamodel. Annotations in the Object Constraint Language (OCL) are used in places for navigation and expression of constraints. *Tagged values* assist in the creation of forall patterns, where additional information about the collection to which a pattern applies needs to be specified. On the RHS, we *stereotype* objects to indicate which elements are modified, added to or removed from the model as a result of the transformation.

In this report, we describe the transformation language in the following manner. We first explain how to construct the LHS of a transformation rule by going through all the necessary concepts from the three lower tiers of the language. The RHS of a rule uses the same building blocks as the LHS, with a few exten-

sions. We introduce these additional language features required for expressing the RHS next. Here we also discuss how information is exchanged between the two sides of a transformation rule. The control flow tier of the language is left for last in our description, as it requires an understanding of the structure of individual rules.

3 Left Hand Side

The LHS of a transformation rule in our language can be composed of three types of patterns: simple patterns, antipatterns and forall patterns. All these patterns are represented by object diagrams, where each object corresponds to a distinct instance of a class from the underlying metamodel. A subset of the source model matches a pattern if there is a one-to-one mapping between their respective elements and relations. Model elements are mapped using the metamodel classes that they instantiate, as well as their attribute values if any are specified in a pattern. The diagram in Figure 2 illustrates the pattern concept.

The object diagram in Figure 2 (a) shows a pattern that consists of a *StructuredActivityNode* named *X*, containing one *Action* and one *ControlFlow*. It can be seen in this diagram that objects are labelled with metamodel class names only, omitting actual object names. Association ends are labelled with *role names* defined in the metamodel. This information is required during matching when more than one association exist between two metamodel classes. Furthermore, role names provide a means of navigation from one object to another by following an association.

Figure 2 (b) shows a source model that contains a match for the pattern in (a), which is shown in bold. The one-to-one mapping between the elements and relations in the pattern and the source model can be clearly seen in this example. It is important to point out that attribute values are used for matching only if these are specified for objects in the pattern diagram. For instance, the *StructuredActivityNode* object in the pattern shown in Figure 2 (a) can only be matched against a *StructuredActivityNode* object in the source model, provided that the *name* attribute of the object in the source model is also assigned to *X*. On the other hand, the *Action* object in the same pattern does not specify any attribute values and can be matched against any *Action* instance in the source model. In fact, the shown source model contains another match for the given pattern where *Action a2* replaces *a1* in the match depicted in bold in the diagram.

3.1 Simple Patterns

The most basic pattern in our transformation language is called a simple pattern, which can be used to check whether certain elements and relations exists in the source model. The simple pattern in Figure 2 (a) can be used to check that a particular transformation rule only applies if the source model includes at least one *StructuredActivityNode* containing at least one *Action* and at least one *ControlFlow*.

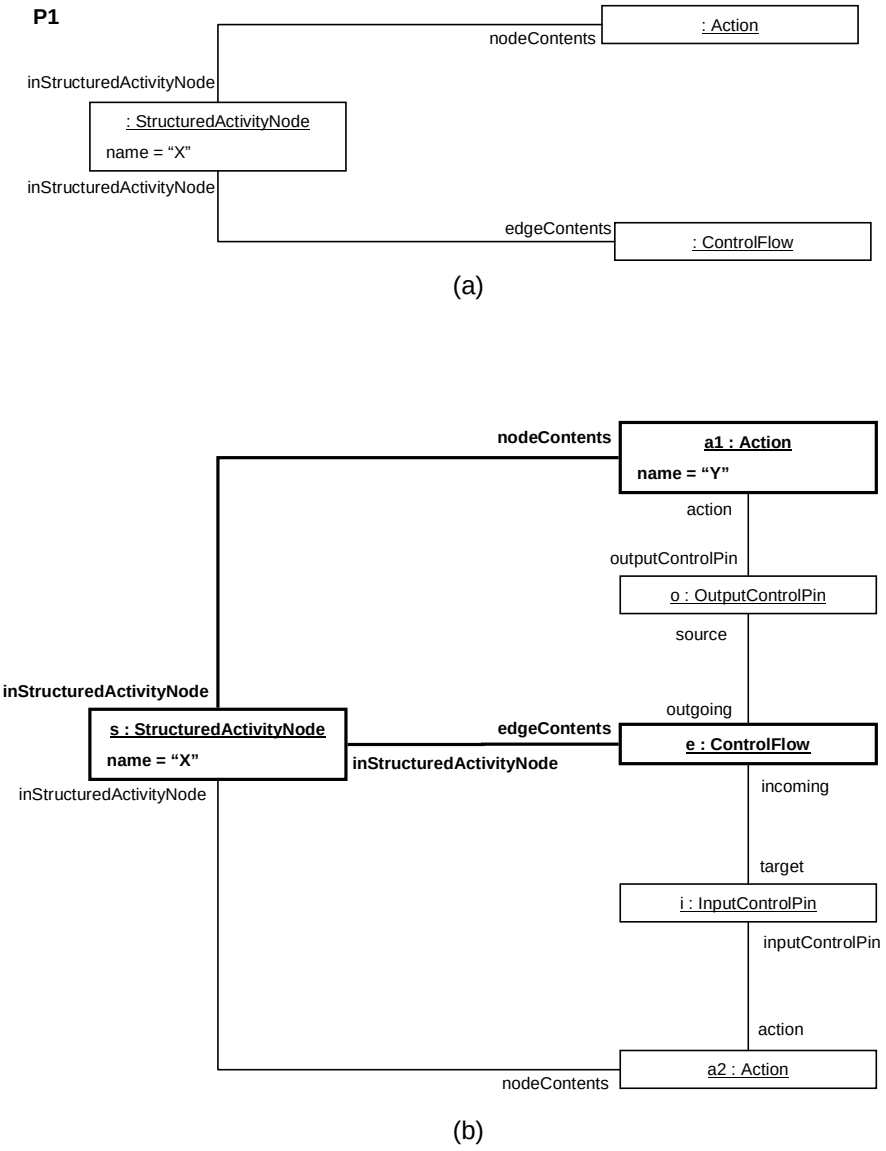

Fig. 2. (a) A Pattern (b) A Match found in a Source Model

3.2 Antipatterns

For certain transformation rules, it is also necessary to check that a particular pattern does not appear in the source model. For instance, we can check that the source model does not have a *StructuredActivityNode* containing two distinct *Action* elements with the pattern shown in Figure 3. Such a pattern that

must not match the source model is called an antipattern in our transformation language.

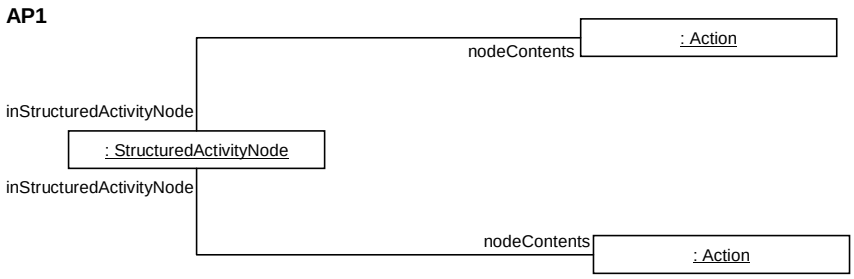

Fig. 3. An Antipattern

The model given in Figure 2 (b) contains a match for the above antipattern, and hence the antipattern is not satisfied by this model. Note that the syntax for simple patterns and antipatterns is identical, but their interpretation as part of the LHS of a transformation rule is different.

Certain universal constraints on the source model can be captured using antipatterns. For example, suppose that we wanted to ensure that every *Action* in the source model leads to at most one other *Action* via *ControlFlow*. Naturally, this constraint does not hold in any model that contains an *Action* leading to two or more other *Actions*. The antipattern in Figure 4 represents the counterexample for this universal constraint and it is sufficient for determining whether the constraint holds.

3.3 Composition of Patterns and Object Sharing

In general, the LHS of a transformation rule is too complex to be expressed by a single simple pattern or antipattern - a composition of these is required. Our transformation language allows for composition of patterns with *disjunction* (`OR`) and *conjunction* (`AND`). In order to distinguish between simple patterns and antipatterns in pattern composition, antipatterns are *negated* (`NOT`). Each pattern is given a name that is unique within the scope of the given transformation. The name of the simple pattern in Figure 2 (a) is `P1` and the antipattern in Figure 3 is `AP1`. Two definitions of the LHS for a transformation rule are possible using these patterns: `P1 OR NOT AP1` and `P1 AND NOT AP1`. The source model in Figure 2 (b) satisfies the former, but not the latter of these two definitions.

In the above example, `P1` and `AP1` are independent from each other. In other words, we can match each one against the source model individually and then combine the results of the matching with either disjunction or conjunction. However, it is often desirable to have multiple patterns refer to the same object in the source model. Such *object sharing* between patterns is supported through common object names in pattern diagrams. Both `P1` and `AP1` include an instance

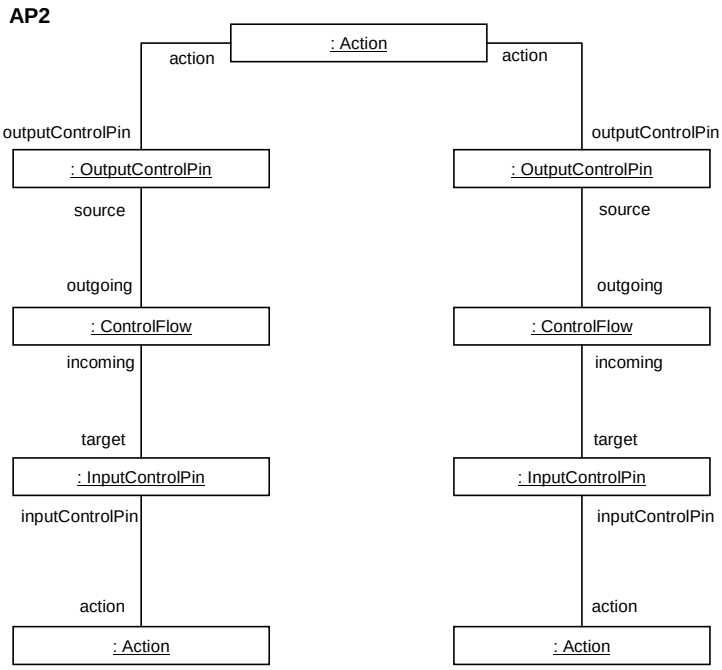

Fig. 4. An Antipattern as a Counterexample for a Universal Constraint

of the *StructuredActivityNode* class. By assigning the same class instance name to both of these objects, we can indicate that the patterns refer to the same element in the source model. Unnamed objects in a pattern are called *free*, while those that are assigned names for object sharing between patterns, are called *bound*.

Consider the simple patterns shown in Figure 5. `SP1` and `SP2` contain only free objects, and `SP1 AND SP2` is satisfied in the source model in Figure 2 (b). On the other hand, `SP3` and `SP4` share an object of type *Action* labelled *a* in the diagrams. The introduction of bound objects changes the meaning of the conjunction of the two patterns, and `SP3 AND SP4` is no longer satisfied in the same source model in Figure 2 (b).

Composition of patterns and object sharing are concerns of the transformation rules tier in our language. In fact, the LHS of any transformation rule is a composition of simple patterns, antipatterns and forall patterns, which are described next.

3.4 Forall Patterns

Composition of simple patterns and antipatterns is still not expressive enough for capturing certain universal constraints. For instance, a counterexample antipattern cannot be used for checking that all *Actions* in a model are contained in

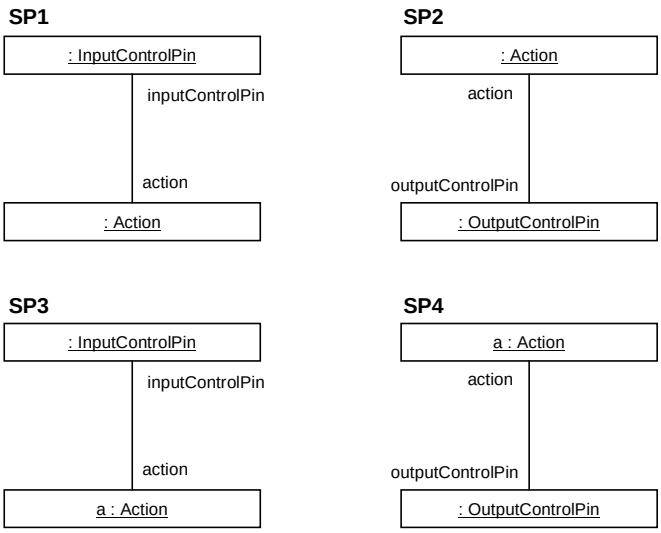

Fig. 5. Free and Bound Objects in Patterns

some *StructuredActivityNode*. We use forall patterns that are based on Milicev's *ForEach* packages [Mil02b], to show that a particular pattern or composition of patterns must apply to multiple objects of the same type within the source model. The forall pattern for the above-mentioned constraint is shown in Figure 6.

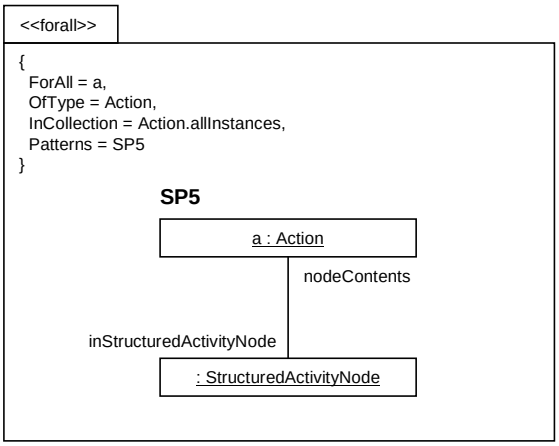

Fig. 6. A Forall Pattern

As can be seen in the diagram in Figure 6, a package with a `forall` stereotype is used to show the scope of a forall pattern. The tagged values in curly brackets indicate the collection to which the patterns contained in the package must apply. In this example, the simple pattern `SP1` must apply for all elements *a*, of type *Action*, that are contained in the collection of all instances of the *Action* class in the model. More generally, the value of the *ForAll* tag is the name of the bound object that refers to individual elements in the collection under consideration. This bound object must appear in at least one of the patterns contained in the forall pattern. The tagged value *OfType* simply indicates the type of the elements in the collection. Navigation to the collection is done using OCL expressions in the value of the *InCollection* tag. The *allInstances* operation is predefined in OCL and can be used on a class to navigate to all objects in a model that instantiate that class. The *Patterns* tagged value states the patterns contained inside the forall pattern. It can be assigned to a single pattern or a composition of patterns. The diagrams for the contained patterns can be drawn inside the forall pattern package as in this example, or elsewhere.

A more complex example of using forall patterns with object sharing is presented in Figure 7. Suppose that the LHS of a particular transformation rule is composed of a conjunction of the simple pattern `SP6`, antipattern `AP3` and the forall pattern given in the diagram. `SP6 AND NOT AP3` checks that there exists an *OutputPinSet* in the model that contains *OutputControlPins* but not *OutputObjectPins*. The forall pattern navigates to a collection of *Actions* that can be reached by the *OutputControlPins* inside the *OutputPinSet* found in `SP6 AND NOT AP3`. `SP7` states that each *Action* inside this collection must be inside some *StructuredActivityNode*. This *StructuredActivityNode* must not lead to a *FlowFinalNode* via *ControlFlow*, as captured in `AP4`.

The example in Figure 7 once again illustrates the concept of object sharing. Patterns `SP6` and `AP3` share an object of type *OuptutPinSet* that is labelled *p* in the pattern diagrams. In fact, this particular object is also used in the forall pattern for collection navigation and hence all three patterns are connected through object *p*.

The OCL navigation expression assigned to the *InCollection* tag of the forall pattern in Figure 7 has the following interpretation. All the *ConnectableNodes* that can be reached from the *OutputPinSet p* via *ControlFlow* are gathered with the expression `p.outputControlPin.outgoing.target`. The *ConnectableNodes* class is abstract, having subclasses *ControlNode* and *Pin* where *ControlNodes* comprise *InitialNode* and *FinalNode* (see the BOM specification [FS04] for more details). The expression `select (x | x.oclIsTypeOf(InputControlPin))` chooses only those objects from the collection of *ConnectableNodes* that are instances of the *InputControlPin* class. Finally, navigation to all the *Actions* associated with the *InputControlPins* is done with `.action`.

The simple pattern `SP2` contains an *Action* object named *a* that refers to the individual elements in the collection underlying the forall pattern. In turn,

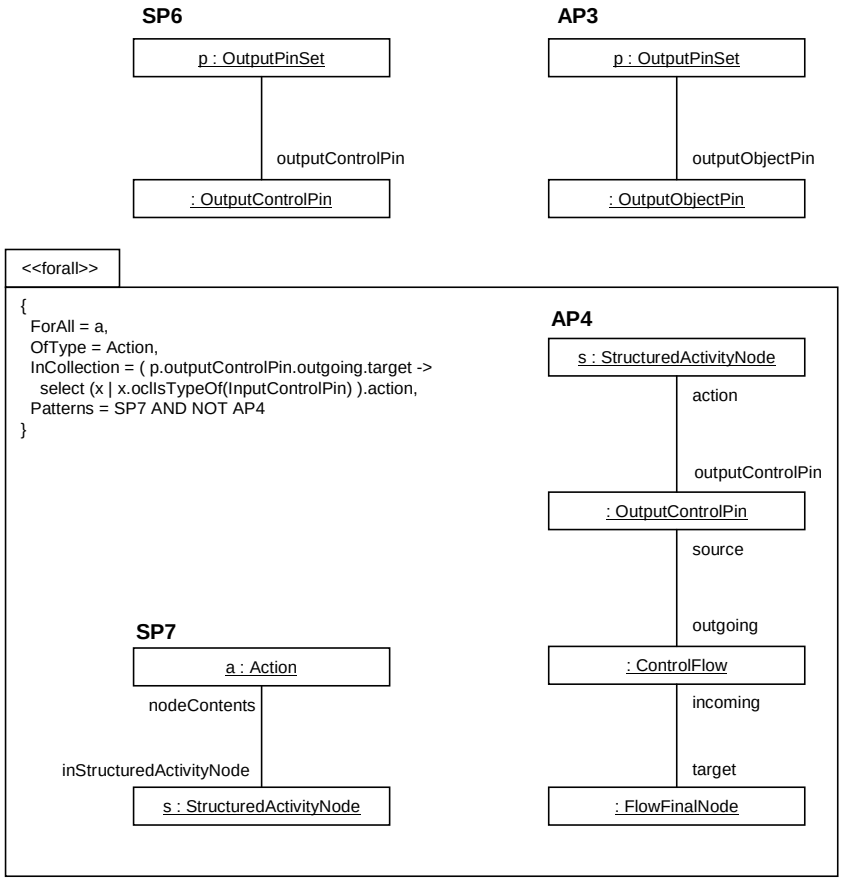

Fig. 7. A Forall Pattern and Object Sharing

`SP2` and the antipattern `AP2` are connected through the bound object s of type *StructuredActivityNode*.

Usually a number of alternative ways of constructing patterns to express the same constraint is available to the user. In our previous example in Figure 7, the forall pattern can be substituted by the one shown in Figure 8. This forall pattern uses a simpler OCL expression for the navigation. However, there are now three patterns instead of two contained in the forall pattern. The composition `NOT AP5 OR (SP8 AND NOT AP6)` applied to all *OutputControlPins* of p, states that either each *OutputControlPin* does not lead to an *InputControlPin*, or it leads to an *Action* inside some *StructuredActivityNode* that is not connected to a *FlowFinalNode* via *ControlFlow*.

Forall patterns can be used to show a variety of constraints on associations between objects. Figure 9 illustrates several of the different ways in which objects inside forall patterns can be associated to other objects in the model.

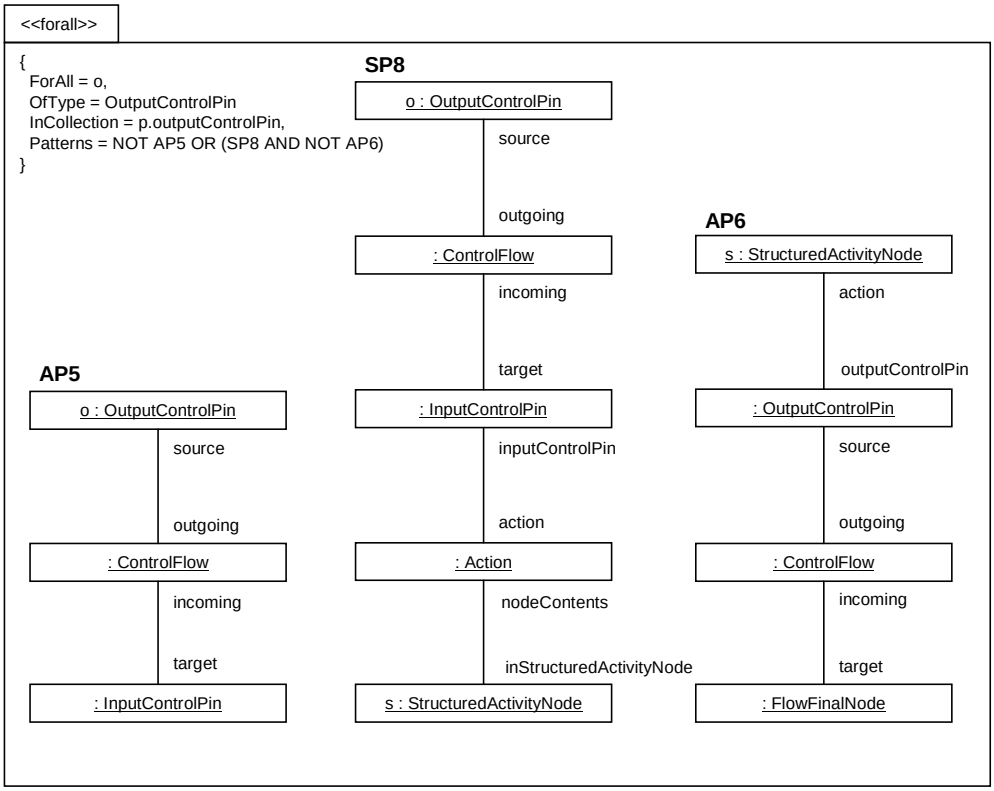

Fig. 8. An Alternative Forall Pattern

In the examples of forall patterns thus far we have only used associations as shown in Figure 9 (a), where the forall pattern applied to a collection of objects of type A and object a referred to individual objects within that collection. Then we interpreted the body of such a forall pattern in a way that every object a in the collection was to be associated with some object b of type B. There is however one subtle detail of this interpretation that has not been finalised in our language. Is there a distinct object b for each object a in the collection or can the associations between objects a and object b overlap? The multiplicity of the association between classes A and B on the metamodel level is one factor that influences the interpretation of such a pattern. If the association between classes A and B is one-to-one in the metamodel, then the pattern in Figure 9 (a) requires that there be a distinct object b for each a.

The other patterns depicted in Figure 9 illustrate further difficulties in defining semantics for forall patterns. For most cases where forall patterns are used, several interpretations could be defined. Further research and case studies are required to determine which of the possible interpretations would be the most appropriate in each case. Moreover, it is not clear whether a semantics for forall

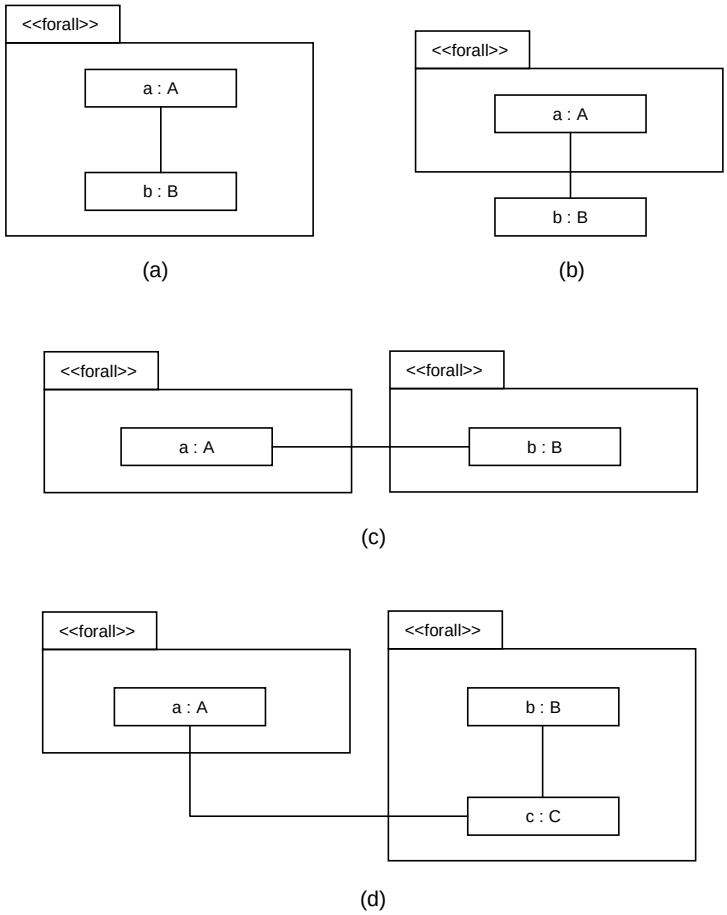

Fig. 9. Associations in Forall Patterns

patterns can be defined in such a way that every possible usage of these patterns is taken into account.

3.5 Object Constraint Language

So far we have only used OCL for navigation purposes in forall patterns. Additionally, it is used in our language to express certain constraints that cannot be captured using patterns. For instance, consider an object *o* of type *LiteralInteger* with an attribute called *value* of type *Integer*. Patterns cannot show that *o.value.intValue()* must be less than 5. On the other hand, this constraint can be easily stated in OCL as an invariant: `context o inv: value.intValue() < 5`.

Potentially, any constraint expressed using a composition of patterns can also be written in OCL. The forall patterns in Figures 7 and 8 in the previous section are equivalent to the following OCL invariant.

```
context p inv:
(p.outputControlPin.outgoing.target -> select (x |
  x.oclIsTypeOf(InputControlPin)
).action -> forAll (a |
  (a.inStructuredActivityNode -> notEmpty()) and
  (a.inStructuredActivityNode -> !exists (s |
    s.outputControlPin.ougoing.target.oclIsTypeOf(FlowFinalNode)))
```

While OCL can be used to completely replace patterns on the LHS of a transformation rule, we recommend its use only in cases where patterns are inapplicable or inconvenient. In this way, OCL constraints can appear as annotations to certain pattern diagrams.

The main strength of OCL is its expressive power, which seems to supersede that of pattern composition. This has already been demonstrated by an earlier example in used in the first paragraph of this section. Furthermore, certain complex constraints that can be expressed using patterns require a large number of pattern diagrams to be drawn. OCL representation for such constraints is usually more compact.

Despite its expressive power, the textual notation prescribed by OCL is not intuitive and requires more expert skills from the user. An OCL constraint that spans several lines is already difficult to comprehend. On the other hand, patterns allow one to build complex constraints by composing small manageable parts - simple patterns, antipatterns and forall patterns. This also allows reuse of patterns in different transformation rules. OCL constraints are not as modular, although they also allow for reuse.

We have now introduced all the concepts required to express the LHS of a transformation rule. Next we discuss some alternative ways of supporting the language features necessary to capture a rule's LHS, as well as possible extensions to the language introduced thus far.

3.6 Possible Extensions and Alternatives

Some of the possible extensions and alternative ways of describing the LHS of a transformation rule are given below.

Null objects. Object diagrams could be extended with *null objects* to show that a certain object does not exist. For instance, this would allow one to construct an antipattern shown in Figure 10 that is equivalent to the forall pattern in Figure 6.

Input and output ports. Implementation of our language in its present form would require unification to take place to resolve object sharing between patterns, as it is done implicitly. Alternatively, *input* and *output ports* can be

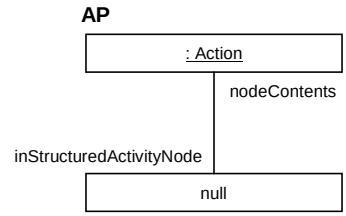

Fig. 10. A Null Object

defined for each pattern to provide explicit passing of objects between patterns [AKL03,Agr04].

Pattern templates. It has already been mentioned that reuse of patterns is possible in different transformation rules. Patterns can be made even more reusable if they are parameterised to create *pattern templates*. An example of this concept is illustrated in Figure 11.

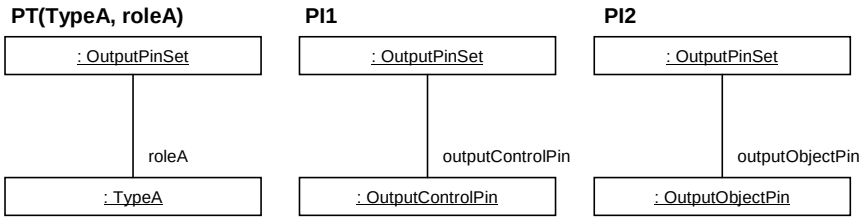

Fig. 11. A Pattern Template

In the diagram above, `PT` is a pattern template with two parameters `TypeA` and `roleA`. The patterns `PI1` and `PI2` are examples of instances of this template. The instance `PI1` is created as follows: `PI1 = PT(OutputControlPin, outputControlPin)`.

Multiobjects. In our current approach, we use forall patterns to deal with collections of objects of the same type. UML *multiobjects* (see e.g. [KHE03]) could be used for similar purposes. Figure 12 shows some of the association constraints that can be expressed using multiobjects.

In diagram (a) above, there is a single object a of type A that must be associated with each object of type B in the collection represented by the multiobject b. In (b), a represents a collection of objects of type A and each individual object in that collection must be associated with an object of type B from collection b. We foresee that the definition of semantics for constraints expressed using multiobjects would be complicated with the same difficulties encountered with forall patterns that were described with reference to Figure 9. The precise relation between forall patterns and multiobjects still needs to be

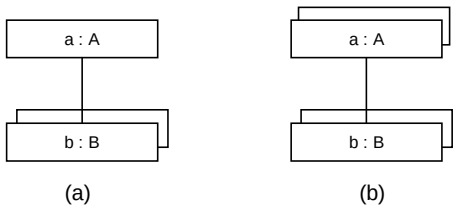

Fig. 12. Multiobjects

explored.

The following section describes the additional features of our language necessary to express the RHS of a transformation rule.

4 Right Hand Side

The RHS of a transformation rule describes the parts of the model affected by the transformation. The same object diagram notation used to express the LHS of a rule is used for the RHS, except for several further extensions. UML stereotypes *new* and *removed* are used on objects and associations to indicate creation and removal of model elements during transformations. Attributes for new objects can be derived from the object attributes in the source model. An example of a rudimentary transformation rule consisting of a LHS and a RHS is given in Figure 13.

The LHS in this example consists of one simple pattern that checks that the source model contains two *Actions* connected with *ControlFlow*. The RHS of the rule also consists of one simple pattern that shows that the transformation replaces the *ControlFlow* between the two *Actions* by *ObjectFlow*. Object sharing between LHS and RHS is done in the same way as between patterns, using instance names of objects. All objects and associations eliminated from the source model are marked with the *removed* stereotype, while those that are created are stereotyped *new*. Objects without stereotypes correspond to the objects from the source model that are preserved by the transformation and thus are part of the target model. Those objects that do not appear in the diagrams of the RHS are implicitly preserved.

On the RHS of a transformation rule, each removed object must be named and its name must either match an object with the same name and type on the LHS, or additional navigation to it must be provided. For example, if the *ControlFlow* instance named *cf* did not appear in the pattern on the LHS, one of the following OCL expressions could be used for the navigation: `cf = oc.outgoing` or `cf = ic.incoming`. Each removed association must either connect two removed objects, a removed and a preserved object, or two preserved objects. Note that the preserved objects connected to a removed association still need

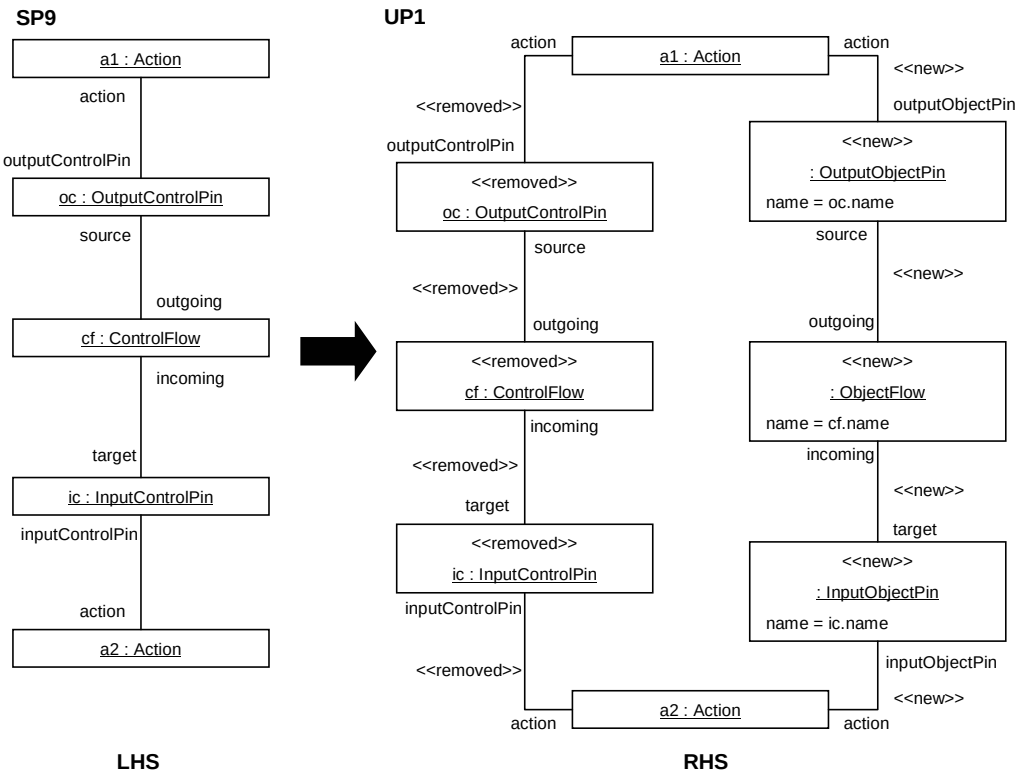

Fig. 13. A Transformation Rule

to appear in the diagram and must be named for the purpose of identifying the removed association.

Attributes for newly created objects can be derived from attributes of source model objects. OCL expressions are used for this purpose, as demonstrated in this example by `name = oc.name` in the new *OutputObjectPin* object. In this case, the name of the removed *OutputControlPin* is simply copied over to the new *OutputObjectPin*.

A transformation may require certain objects to be preserved in the target model, but with modified attributes. In such a case, a *modified* object stereotype should be used in a RHS pattern and new values for modified attributes should be given. Associations from the source model cannot be modified, these can only have a *removed*, *new* or no stereotype.

Figure 14 illustrates how a source model might be updated according to the discussed transformation rule. In this instance, the source model precisely matches the LHS of the transformation rule. Generally however, the LHS is only a subset of all the elements constituting the source model.

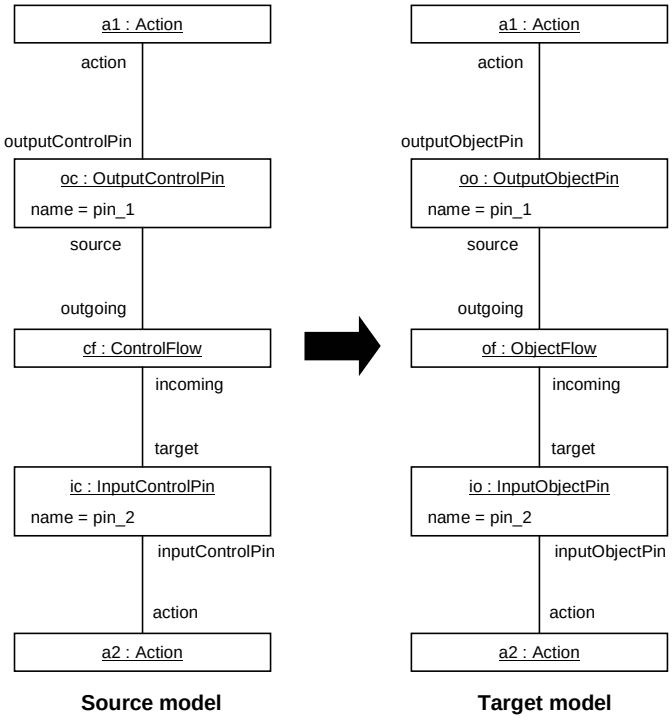

Fig. 14. A Transformed Model

Sets of instances of the same type can also be created or removed by a transformation rule. This is done using forall patterns that were introduced in Section 3.4. The collection underlying a forall pattern on the RHS of a rule must refer to a collection in the source model.

Furthermore, conditional creation and removal of objects and associations can also be expressed on the RHS. Figure 15 depicts an example of conditional creation of an association used inside a forall pattern.

The RHS pattern shown above states that all those *Actions* in the source model that were not inside a *StructuredActivityNode* are placed into a new *StructuredActivityNode* in the target model.

Similarly to the LHS of a transformation rule, the RHS comprises a composition of patterns. However, this composition can only consist of simple and forall patterns joined with conjunction. Antipatterns and disjunction cannot be used on the RHS.

4.1 Possible Extensions and Alternatives

Alternatively to the approach described in this section, the RHS could be expressed in a more imperative manner. The update of the source model could be

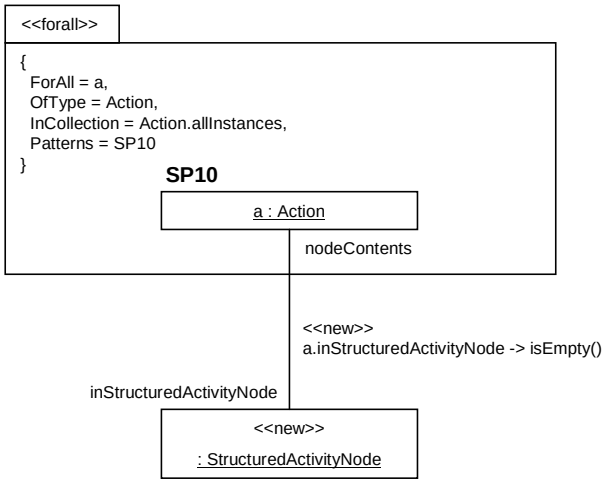

Fig. 15. Conditional Creation

done incrementally rather than captured declaratively. For instance, individual object diagrams could represent steps in the update and one could then specify the sequence in which these steps must be executed.

5 A Transformation Rule

We have now explained how to describe the source model on the LHS of a rule and the target model on the RHS of a rule. Both sides consist of a composition of patterns and a full transformation rule can be given in the following format: `Rule: LHS` → `RHS`. Here is an example of a transformation rule: `RuleA: (SP1 OR SP2 AND NOT AP9)` → `(UP1 AND UP2)`.

It is important to note that on the LHS of a transformation rule, patterns can be composed with conjunction and disjunction. Furthermore, atomic negation is used to indicate antipatterns on the LHS. However, general negation such as `NOT (SP1 AND SP2)` is not supported in pattern composition.

On the RHS of a rule, update patterns can only be composed with conjunction. Allowing the use of disjunction, would result in a non-deterministic update of the source model. Pattern composition on the RHS can consist of simple and forall patterns, but not antipatterns.

6 A Transformation Unit

A transformation unit defines control flow for the entire transformation process that usually involves application of several transformation rules to the original source model. Thus far we have not explored what type of control flow mechanisms are required in a transformation unit. We expect that choices between

transformation rules, as well as sequence and iteration of rules need to be supported by the language at the transformation unit tier.

7 Validation by Case Study

Business process modelling with BOM is supported by the IBM WebSphere Business Integration Modeler, which also allows one to automatically generate implementation code for these models in *Business Process Execution Language for Web Services (BPEL4WS)*. However, there is a mismatch between the semantics of BOM and BPEL4WS with respect to support for cyclic control flow between process activities. BOM allows unstructured cycles to occur in the models, while BPEL4WS only supports structured cyclic control flow in the form of while-loops. The solution to this problem that was investigated in our case study involves removing unstructured cycles in a BOM model before generating code from it. At present, the WebSphere Modeler requires one to resolve unstructured cycles in process models manually as a required step before code generation. As the manual process for this is naturally time-consuming and error-prone, automating this procedure would greatly benefit the user. Our group has designed and implemented a prototype for this transformation, which has now been handed over to the WebSphere Modeler development team.

The main algorithm underlying the transformation for removing unstructured cycles was derived from control flow T1-T2 analysis in compiler theory. While the objective of the original T1-T2 analysis is to determine reducibility of a given flow graph, the BOM transformation had to preserve the behaviour captured in the source model.

The design of the transformation consisted of a set of diagrams, informally showing how different cases should be handled in the implementation. These design diagrams were mainly used for discussions between the developers and as a rather informal and incomplete specification of the transformation on which the implementation was based. Implementation was done in Java using the *Eclipse Modelling Framework (EMF)* to manipulate the internal representation of BOM models required by the transformation. Such an approach to developing a model transformation proved to be time-consuming. Additionally, the resultant transformation implementation could not be easily changed or reused for another similar transformation.

The objective of our case study was to capture the details of the cycle-removal transformation using the pattern-based language described in this report. Using such a model-based representation of the transformation seems to be more suitable for analysis than the Java code [Küs04]. Furthermore, the ultimate goal is to verify the design and then use it to generate the transformation code automatically. Flexibility and reusability of transformations will also be improved with such an approach.

During the case study we concentrated on one part of the cycle-removal transformation, the so-called T2 transformation step. The fundamentals of this transformation step are depicted in the diagram in Figure 16.

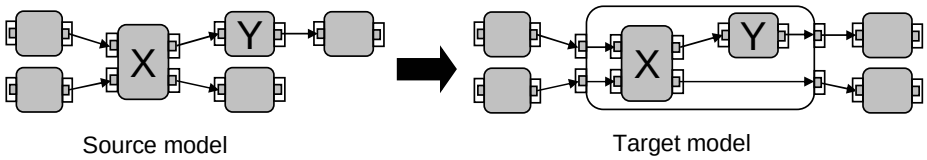

Fig. 16. Transformation Step T2

The left part of the diagram in Figure 16 shows that the transformation applies to a part of the source model containing two *Actions* `X` and `Y`, provided that `X` is the only predecessor of `Y`. During the transformation both `X` and `Y` are placed into a new *StructuredActivityNode*, as shown on the right of Figure 16. Several more complex preconditions for this transformation are not illustrated in Figure 16. For example, all the *Pins* of `X` and `Y` must be connected to an *ActivityEdge*.

Using the Java implementation as reference for the details about the T2 transformation, we created one transformation rule using the presented language. The LHS of the rule consisted of a composition of 29 different patterns and OCL constraints. The RHS of the rule required a conjunction of 14 update patterns. Some simple patterns and antipatterns could be reused within forall patterns on the LHS. In addition there was much superfluous replication during the rule construction and it was apparent that further reuse mechanisms need to be included in the language.

On the whole, the T2 transformation step was successfully expressed using our language. The main benefit of the approach was that it allowed us to construct a complex transformation rule from manageable building blocks. The visual pattern representation made the transformation design more comprehensible, while keeping it precise at the same time. We did however resort to using OCL for some complex constraints where using patterns did not seem feasible. As already mentioned, we felt that more reusability mechanisms in the language would ease the transformation design process even further.

8 Conclusion and Future Work

In this report we presented a layered approach to defining a language for describing model transformations. In many places however, the syntax and semantics of the language still need to be finalised.

Our case study of resolving unstructured cycles in BOM models allowed us to validate the transformation language in its current state. The design process of the T2 transformation showed that most of the language constructs are applicable, but at the same time pointed out areas where the language could be improved. A further case study investigating a transformation across different representations should also be undertaken to validate the language on transformations involving more than one metamodel.

Once the transformation language syntax and semantics are finalised, possible useful analyses of transformations at the design level need to be considered. Verifying a transformation on the design level would allow one to improve the quality of transformations before they are implemented. For instance, even though the meaning of a single pattern is easy to understand, a composition of patterns can be difficult to comprehend. Checking pattern compositions and transformation rules for contradictions is one example of useful analyses that could be performed at the transformation design level.

Finally, a transformation development environment with features allowing the design, analysis and code generation of model transformations needs to be developed to truly demonstrate the value of our proposed approach.

References

[Agr04] Aditya Agrawal. *A Formal Graph Transformation Based Language for Model-to-Model Transformations.* Dissertation for PhD in Electrical Engineering, Faculty of the Graduate School of Vanderbilt University, Nashville, Tennessee, August 2004.

[AKL03] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An End-to-End Domain-Driven Software Development Framework. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 8–15. ACM Press, 2003.

[FS04] Joachim H. Frank and Ghaly Gamil Stefanos. *Business Operations Metamodel (BOM). Draft version 2004-05-21.* IBM, May 2004.

[KHE03] J. M. Küster, R. Heckel, and G. Engels. Defining and Validating Transformations of UML Models. In J. Hosking and P. Cox, editors, *IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003) - Auckland, October 28 - October 31 2003, Auckland, New Zealand, Proceedings*, pages 145–152. IEEE Computer Society, 2003.

[Kus00] S. Kuske. *Transformation Units - A Structuring Principle for Graph Transformation Systems.* Dissertation, Universität Bremen, 2000.

[Küs04] J. M. Küster. Systematic validation of model transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004), Lisbon, Portugal*, October 2004.

[mda02] The Model-Driven Architecture, Guide Version 1.0.1, omg/2003-06-01. OMG Document, April 2002.

[Mil02a] Dragan Milicev. Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments. *IEEE Transactions on Software Engineering*, 28(4):413–431, April 2002.

[Mil02b] Dragan Milicev. Domain Mapping Using Extended UML Object Diagrams. *IEEE Software*, 19(2):90–97, March/April 2002.

[Obj04] Object Management Group (OMG). *QVT-Merge Group. MOF 2.0 Query/Views/Transformations, Revised Submission. OMG Document ad/2004-04-01*, April 2004.