Reusable Model Transformation Patterns

Maria-Eugenia Iacob
University of Twente
m.e.iacob@utwente.nl

Maarten W. A. Steen
Telematica Instituut
maarten.steen@telin.nl

Lex Heerink
Telematica Instituut
lex.heerink@telin.nl

Abstract

This paper is a reflection of our experience with the specification and subsequent execution of model transformations in the QVT Core and Relations languages. Since this technology for executing transformations written in high-level, declarative specification languages is of very recent date, we observe that there is little knowledge available on how to write such declarative model transformations. Consequently, there is a need for a body of knowledge on transformation engineering. With this paper we intend to make an initial contribution to this emerging discipline. Based on our experiences we propose a number of useful design patterns for transformation specification. In addition we provide a method for specifying such transformation patterns in QVT, such that others can add their own patterns to a catalogue and the body of knowledge can grow as experience is built up. Finally, we illustrate how these patterns can be used in the specification of complex transformations.

1. Introduction

OMG's Model-Driven Architecture (MDA) ([9], [6]) has emerged as a new approach for the *design and realisation of software* and has eventually evolved in a collection of standards that raise the level of abstraction at which software solutions are specified. The central idea is that computational independent models (CIMs), platform independent models (PIMs) and platform specific models (PSMs) – defined at different levels of abstraction – are derived (semi-) automatically from each other through *model transformations*. Model transformations are thus a crucial element in OMG's vision on MDA. Transformations relate the different abstractions used in a model-driven development scenario. Model-to-model (M2M) transformations relate CIMs to PIMs and PIMs to PSMs, while Model-to-Text (M2T) transformations relate the PSMs to code. OMG has recently adopted standard languages for the specification of model transformations, for which a number of implementations are already available. The availability of these transformation engines, in addition to the existing metamodelling technology, brings us a lot closer to the realization of the MDA vision. Modelling engineers are now able to define their own Domain-Specific Languages (DSLs) and transformations between them and existing languages.

Since the technology for executing transformations written in high-level declarative specification languages (such as those included in the QVT standard) is of very recent date, we observe that there is very little knowledge available on how to write such declarative model transformations. This led us to the conclusion that there is a need for a body of knowledge concerning the emerging discipline of transformation engineering.

In this paper we aim to make an initial contribution to this emerging discipline. Recently we have had the opportunity to experiment with implementations of both the QVT Core language (from Compuware) and of the QVT Relations language (from IKV++). Based on these experiences we propose a number of useful problem-solution patterns, similar to the well-known design patterns in software development. In addition we provide a method for documenting and specifying such reusable transformation patterns, such that others can add their own patterns and the body of knowledge can grow as experience is built up. For this purpose we have recently started a Wiki catalogue [10] where transformation patterns can be documented and discussed.

The paper is organised as follows. In Section 2 and Section 3 we discuss briefly the QVT model transformation specification standard and a few modelling languages we use in this paper. In Section 4 the issue of documenting transformation patterns is addressed. Section 5 consists of a catalogue of model transformation patterns we believe to be relevant in the context of model-driven development. Each pattern is

described using a template that includes (for illustration purposes) a pattern application example. In Section 6 we demonstrate how the patterns can be used combined by specifying a transformation for state chart models. Finally, Section 7 summarises our conclusions and gives some pointers to future work.

2. The QVT transformation languages

In order for design patterns to be understood and useable by a wide audience, they should be expressed in a well-known, preferably standardized language. QVT (Query/View/Transformation) provides such languages for M2M transformation specification. QVT actually defines three different transformation languages: *Relations, Core* and *Operational Mappings*. Relations and Core are both declarative languages at two different levels of abstraction, with a mapping between them. We briefly present the Relation language below that has been used for specification purposes throughout this paper. For a complete definition of these languages we refer the interested reader to the standard specifications [8].

The QVT Operational language extends both Relations and Core and provides a way of specifying transformations imperatively. As we focus on declarative transformation specification, we will not discuss the Operational language further in this paper. OMG has recently also approved the MOF Model-to-Text standard for specifying transformations from MOF models to text (i.e., code). However, M2T transformations are of a completely different nature and therefore also fall outside the scope of this paper.

In the **QVT Relations** language transformations are specified by defining the relations that should hold between source and target domains. Transformation rules are described in terms of relations that define a mapping between source and target elements and can be constrained in the when and where clauses. Only model elements that satisfy the constraints will be related. Such constraints typically deal with the properties of the model element, such as attributes and associations to other elements. The when-clause specifies a precondition. Only when all conditions in this clause evaluate to true the relation between the specified domains is established. The where-clause specifies a postcondition. Once the relation is established then the conditions specified here should be enforced to hold. When a domain is marked as **enforced**, the engine may create or update that domain in order to establish the relation.

3. Modelling languages

Before addressing the main topic of this paper - the *transformation rule patterns* - we briefly describe the experimental setting in which our results have been devised. The following modelling languages that have served as source and target languages in our transformation pattern specifications:

The **shape language** is a simple, purely syntactical language that has been defined in order to illustrate the model transformation patterns. It does not require any prior knowledge and it basically has only two concepts: simple shape and arrow. There are three types of simple shapes: square, circle and triangle. Furthermore, the Shape language contains a grouping concept called Block used to express hierarchy. A block may contain simple shapes and other blocks. Each shape model should have a unique root element, which is an instance of RootBlock, a specialization of Block. To express relations between simple shapes and blocks the Arrow concept is used. The Shape metamodel is given in the Figure 1.

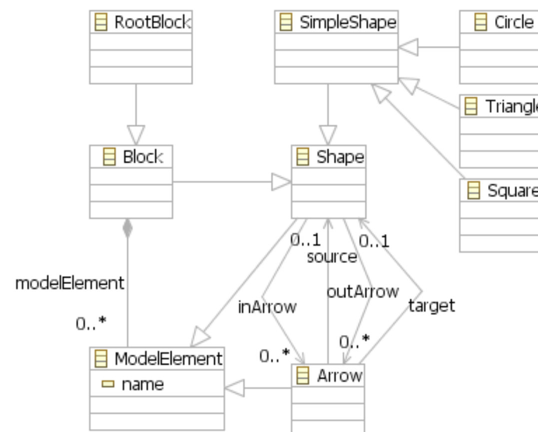


Figure 1. Shape language metamodel

In order to illustrate the transformation patterns proposed in this paper we have used well known diagramming notations, namely the **UML class, activity and statechart diagrams** (for the complete specifications see [7]). The used statechart metamodel can be found in the Figure 2.

4. Transformation design patterns

Since the publication of "Design Patterns" by Gamma et al. [4], patterns are well known in software engineering. Patterns describe which problems software engineers can encounter, the context in which such problems may appear, and a general solution to them. Analogously we propose to start a collection of reusable design patterns for specifying model

transformation. A transformation design pattern, or *transformation pattern* for short, is then a reusable solution to a general model transformation problem.

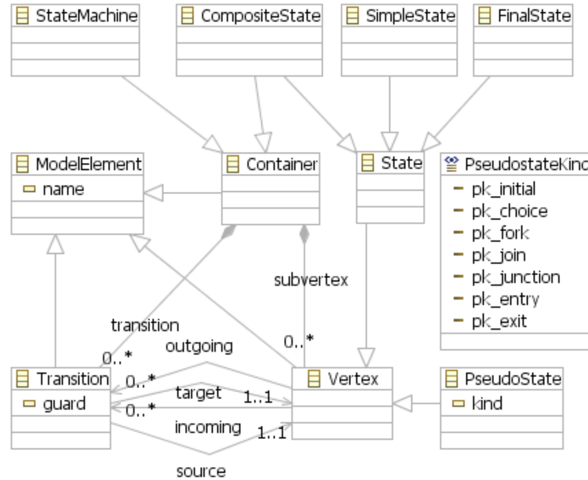


Figure 2. Statechart diagram metamodel

The need for transformation patterns emerged almost immediately after we first started writing model transformations in QVT. Transformations are often very similar. An existing transformation specification is often a good starting point for a new one. Unfortunately, the collection of existing and well-documented transformations is still very small. We also noticed that the same transformation can often be specified in subtly different ways. On the surface it seems to be just a matter of style, but such a different 'style' can have great consequences for performance, applicability and reusability of the transformation. Finally, our first solution to a particular transformation problem often was not entirely correct and had to be revised several times. A library of reusable transformation patterns should enable engineers to get it right more quickly.

4.1. Transformations, transformation rules and rule patterns

Transformations, transformation rules, transformation patterns, rule patterns are just a few concepts which are used with different meanings and sometime interchangeably in the literature. For example, in [5] a definition is given for a transformation pattern which corresponds to what we call a transformation definition. Somewhat similar definitions to the ones we propose are given by [3]. Another interesting view on transformation patterns is that taken by the project Modelware [1] that considers that a transformation pattern (as general repeatable solution to a commonly-occurring model transformation design problem) is not a finished design that can be transformed directly into a transformation specification. Although [1] proposes a catalogue of transformation patterns, their approach is different in two respects. Firstly, Modelware does not rely on the QVT standard for the specification of proposed patterns. Instead, the hybrid imperative/declarative ATL language is used. Secondly, the patterns included in [1] do not overlap with those proposed in this paper.

Therefore, before discussing specific transformation patterns as mentioned before, we feel compelled to provide further clarifications concerning the semantics we have attributed to these concepts and the relations between them (as depicted in Figure 3). In the remainder of this paper the following definitions will be used.

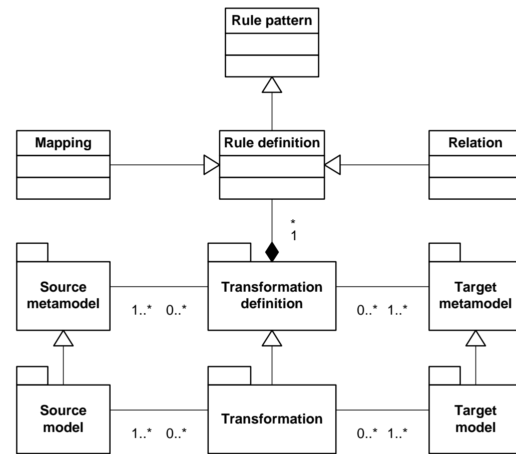


Figure 3. Model transformation concepts and relations between them

A *transformation definition* is a formal specification that consists of a set of rule definitions. A *rule definition* is a formal specification in the form of a *mapping* (in the sense of the QVT - Core language) or of a *relation* (in the sense of the QVT - Relations language). In its simplest form (and in line with the MDA), a *model transformation* is the process of converting a *source model* that conforms to a *source metamodel* into a target model that conforms to a *target metamodel*, using an existing transformation definition between the two metamodels. When a source model is transformed into the target model the transformation definition prescribes the manner in which the different rule definitions that are included in the transformation definition are "executed". In this paper we argue that rule definitions can be created by instantiating so called rule patterns. More specifically, we regard a *rule pattern* as a generic (possibly parameterized) formal specification that describes at a higher level of abstraction a whole class of recurring rule definitions.

4.2. Documenting transformation patterns

A design pattern names, abstracts, and identifies the key aspects of a common design structure, such that it can be reused and applied over and over again in creating new designs. According to [1] and [4], a pattern description should contain the following four essential elements: the pattern name, a description of the problem and the contexts in which it is applicable, the solution to the problem, and the consequences of using the pattern. In addition, pattern descriptions should provide an example to clarify the provided solution.
Likewise, we use a fixed template for documenting the transformation patterns, consisting of the following elements: the **name** of the pattern, the **goal** of the pattern, **motivation** for the pattern, describing the class of problems that the pattern solves, **specification** of the solution using the QVT Relations language, an **example** in which the pattern is applied and considerations regarding the pattern's **applicability**.

5. A catalogue of rule patterns

In this section, we document a number of transformation patterns using the template described above. These are: Mapping, Refinement, Abstraction, Duality and Flattening.

5.1. The Mapping pattern

Goal: Establish one-to-one relations between elements from the source model and elements from the target model.
Motivation: Mapping is the most common and straightforward transformation problem. It occurs when source and target models use different languages or syntax, but otherwise express more or less the same semantics. This pattern is used to a greater or lesser extent in virtually any transformation.
This is the most basic transformation pattern. Typical examples of transformation rules that are based on this pattern are 1-to-1 model transformation rules. It is in general bidirectional (unless different concepts from the left domain are mapped onto the same concept in the right domain). All other transformation patterns use/include this pattern.
Specification:

```
top relation XYMapping {
	nm: String;
	enforce domain left x: X {
	context = c1 : XContext {},
		name = nm };
	enforce domain right y: Y {
		context = c2 : YContext {},
		name = nm };
	when {
		ContextMapping(c1,c2);
	}
}
```

This rule specifies that some element x of type X is related to some element y of type Y, whenever their respective contexts are related by ContextMapping and their names are equal. When the respective model elements have more properties than a context and a name, these should also be mapped. Consider for example the case where the model elements to be mapped represent associations or relationships between other model elements, their sources and targets. The pattern for this case is specified below:

```
top relation RelationshipMapping {
	nm: String;
	enforce domain left a: A {
		context = c1 : AContext {},
		name = nm,
		source = as : AS {},
		target = at : AT {}
	};
	enforce domain right b: B  {
		context = c2 : BContext {},
		name = nm,
		source = bs : BS {},
		target = bt : BT {}
	};
	when {
		ContextMapping(c1,c2);
		ElementMapping(as,bs);
		ElementMapping(at,bt);
	}
}
```

Example: For an example of mapping pattern instance one may refer to the `relation TransitionMapping` in Section 6. Besides, we have applied this pattern to relate Circles to Squares in the Shape language. The complete specification of this transformation can be downloaded from our Wiki catalogue [10].
Applicability: The mapping pattern can be used to:

- translate a model from one syntax into another syntax, e.g. from ecore to XML, or from UML to Java;
- relate concepts one-to-one in source and target model.

5.2. The Refinement pattern

The refinement pattern is the key design pattern in stepwise refinement, which is a method to create lower level (or: concrete) models from models from higher level (or: abstract) models in a number of successive refinement steps. Refinement is a key ingredient of MDA, which advocates the realization of software

systems through systematic stepwise refinement from models. Depending on the subject, different refinement types can be distinguished, e.g., relation refinement and node refinement.

Relation refinement pattern

Goal: To obtain a more detailed target model by refining an edge to multiple, possibly interrelated, edges.
Motivation: Relation refinement is typically used to detail steps (which are often modelled as edges) into sub steps. An example is e.g., by adding process steps to an existing UML activity diagram.
Specification: In relation refinement an edge is refined to (a set of) edges, possibly interleaved with nodes. The corresponding pattern is characterized by a single relation mapping on the left and multiple relation and/or node mapping on the right. The pattern for relation refinement is straightforward, and closely resembles the Mapping Pattern. The specification below demonstrates the mapping of an edge e1 to an edge-node-edge pattern.

```
top relation RelationRefinementMapping {
        n : String;
        enforce domain left e1 : Edge {
                name = n,
                context = c1 : Context {},
                source = s_left : Node {},
                target = t_left : Node {}
        };
        enforce domain right im_node {
                context = c2 : Context {}
        -- an intermediate node
                };
-- potentially more nodes and edges
        enforce domain right e2 : Edge {
                source = s_right : Node {},
                name = s_right.name + '_to_' +
im_node.name,
                context = c2,
                target = im_node
        };
        enforce domain right e3 : Edge {
                target = t_right : Node {},
                name = im_node.name + '_to_' +
t_right.name,
                block = c2,
                source = im_node
        };
        when {
                ContextMapping(c1,c2);
                ElementMapping(s_left,s_right);
                ElementMapping(t_left,t_right);
        }
}
}
```

Example: An example of relation refinement in the Shape language is the refinement of any Arrow into an Arrow-Square-Arrow combination. The corresponding specification in QVT Relations can be downloaded from our Wiki catalogue [10].

Node refinement pattern

To obtain a more detailed target model by refining a node to multiple, possibly interrelated, nodes a node refinement pattern (similar to the relation pattern) has been documented. However due to space limitations has not been included in this paper, but can be found on our Wiki catalogue [10]. Node refinement is used to provide more detail to a node. For example, an UML class diagram that leaves the methods and attributes unspecified can be refined to class diagrams that do specify methods and attributes. Another example is to refine a super state in a hierarchical statechart to several interrelated sub-states.

5.3. The Node Abstraction pattern

Goal: Abstracts from nodes in the source model while keeping the incidence relations of these nodes.
Motivation: The node abstraction pattern removes specific nodes from the source model to create a target model whilst preserving the incidence relations. The node abstraction pattern can be used to abstract from specific information from models. The specification below shows a simplified node abstraction pattern that abstracts from a node X and produces an edge between the incidences. It is assumed that source and target have the same metamodel, that node X is a subtype of the abstract type Node, that each node contains references to its incidence edges, and that each edge contains references to its source and target nodes. The pattern below can only handle sequence of X of length 1, multiple in-sequence occurrences of X cannot be handled.
Specification:

```
top relation Node_X_Abstraction {
        enforce domain left s1 : X {
                inEdge = e_in : Edge {
                        name = na_in : String;,
                        source = ss1 : Node {}
                },
                outEdge = a_out : Edge {
                        name = na_out,
                        target = tt1 : Node{}
                }
        };
        enforce domain right a : Node {
                name = na_in + na_out,
                source = ss2 : Node {},
                target = tt2 : Node {}
        };

        when {
                NodeMapping(ss1,ss2);
                NodeMapping(tt1,tt2);
```

```
        }
}
```

Example As node abstraction is quite intuitive we do not provide a code fragment of node abstraction. However, specification of example transformations can be downloaded from our Wiki catalogue [10].
Applicability: Remove model elements from models, for example, remove processes that conform to certain criteria from a process diagram.

5.4. The Duality pattern

Goal: Given a model, to generate its semantic dual.
Motivation: Various modelling languages exist that rely on the (acyclic) directed graph formalism to represent dynamic behaviour (e.g., Petri nets, BPMN and UML statechart diagrams, sequence diagrams, collaborations diagrams and activity diagrams). Nevertheless the semantics attributed to nodes and arrows in these graph-like models differs. There are roughly two main categories of such languages:

- languages that focus on modelling the procedural flow of activities that make up a larger activity, namely a process - in this case vertices generally represent (branching, assembling) activities, while arrows depict causality relations between activities (e.g., BPMN, UML activity diagrams);
- languages that focus on modelling the flow of control from state to state for a particular object undergoing a process - in this case a vertex generally represent one state of that object, while an arrow depict the transition from one state to the other (i.e., indicating that the object being in the first state will enter the second state as a result of reacting to discrete events; e.g., Petri nets, UML statechart diagrams).

Defining transformations between modelling languages that belong to these two different categories requires the application of what we will refer to as *duality pattern* (explained in more detail in the sequel). This pattern is based on the dual character of these two types of languages. More specifically, an activity (in the sense of the first category of languages) can be seen as the procedure that leads to a state change of the object(s) undergoing a process, that is a transition in the sense of the second type of languages, while a causality relationship may be interpreted as the moment when the object(s) have reached a certain state as a result of an activity's completion, which makes possible the initiation of the subsequent one(s). In other words, the duality rule pattern will map vertices from the source model onto arrows in the target model and arrows from the source model onto vertices in the target model. However, it should be noted that the mapping of branching/assembling nodes deserves special consideration.
Specification: Our transformation strategy is as follows: All Arrows on the left are related Nodes on the right using the mapping rule pattern, as indicated below:

```
top relation ArrowNodeMapping {
        nm: String;
        enforce domain left a: Arrow {
                context = c1: AContext {},
                name=nm
        };
        enforce domain right v: Vertex {
                context = c2: VContext {},
                name=nm
        };
        when {
                ContextMapping(c1, c2);
        }
}
```

Rules must be defined for relating a node on the left with one or more arrows on the right for each of the following cases:

- a node on the left, having an incoming arrow e1 and an outgoing arrow e2, is related to an arrow a on the right if e1 has been related to the source of a and e2 to the target of a.

```
top relation NodeArrowMapping {
        nm: String;
        enforce domain left v:Vertex {
                context = c1: NContext{},
                incoming = e1: Arrow {},
                outgoing = e2: Arrow {},
                name = nm
        };
        enforce domain right a:Arrow {
                context = c2: AContext {},
                source = v1: Vertex {},
                target = v2: Vertex {},
                name = nm
        };
        when {
                ContextMapping(c1, c2);
                v.outgoing->size()=1;
                v.incoming->size()=1;
                ArrowNodeMapping(e1, v1);
                ArrowNodeMapping(e2, v2);
        }
}
```

- a node on the left that has an incoming arrow e1 and *n (n>1)* outgoing arrows (i.e., the node is a "split node") will be mapped on *n* arrows on the right (one for each outgoing arrow on the left) using the rule indicated below. As in the case of the previous rule, the rule fires when contexts have been related and the incoming arrow e1 has been related to the source of the arrow a (on the right) and an outgoing arrow e2 to the target of a.

```
top relation SplitArrowMapping {
        nm, nm2: String;
        enforce domain left e2:Arrow {
                source = v:SplitNode {
                        context = c1: SContext
{},
                        incoming = e1: Arrow
{},
                        name = nm
                },
                name = nm2
        };

        enforce domain right a: Arrow {
                context = c2: AContext {},
                source = v1: Vertex {},
                target = v2: Vertext {},
                name = nm.concat(nm2)
        };

        when {
                ContextMapping(c1,c2);
                v.outgoing->size()>1;
                v.incoming->size()=1;
                ArrowVertexMapping(e1, v1);
                ArrowVertexMapping(e2, v2);
        }
}
```

- a similar rules can be defined when the node on the left is a “join node”;
- rules must also be defined when the node on the left is a start node/final node (no incoming/outgoing arrows) or the node is simultaneously join and split node (two or more incoming arrows and two or more outgoing arrows). Because of space limitations we do not provide the specification of these rules, although for a complete transformation these situations must be equally considered.

Example: An example duality pattern application is the generation of a statechart diagram from an activity diagram. The corresponding QVT specification can be downloaded from our Wiki catalogue [10].
Applicability: The duality pattern can be used to related models expressed in languages between which a duality relationship can be established (i.e., nodes/constructs from the source language can be semantically related/mapped to arrows/relations in the target language and, relations/arrows in the source language can be related/mapped to nodes/constructs in the target language). For example it can be used to define transformations between UML activity diagrams and UML statechart diagrams. It should be noted, that situations may occur (depending on the metamodels of the involved languages) when this type of pattern is not bidirectional.

5.5. The Flattening pattern

Goal: Remove the hierarchy from the source model.
Motivation: Models are often hierarchically structured. Consider for example package hierarchy in UML, composite states in Statecharts or Hierarchical PetriNets. Such hierarchical structuring usually is intended to make the models easier to understand and do not have inherent semantics. In order to realize such hierarchical models in code or formally analyze them using some tool, it may be necessary to first flatten the model to a model without hierarchy.
Specification: We make the following assumptions:

- Source and target models have the same metamodel.
- Source and target models both have a unique RootElement, which are related by the RootMapping relation, an instance of the mapping pattern.
- Model elements in the source model belong to (have as their context) the RootElement or to a Composite element, representing the hierarchy.

Our transformation strategy is as follows. All Composites on the left are related to the RootElement on the right. The CompositeContext here is either the RootElement or another Composite. Thus the CompositeContext c1 should be related to the RootElement r via RootMapping or CompositeFlattening itself.

```
top relation CompositeFlattening {
        checkonly domain left c: Composite {
        context = c1 : CompositeContext {} };
        enforce domain right r: RootElement{};
                when {
                RootMapping(c1,r) or
CompositeFlattening(c1,r);
        }
}
```

All other elements will be simply copied using instances of the mapping pattern above. In these rules the ContextMapping should be replaced by the when clause of the CompositeFlattening rule.

```
relation ElementMapping {
        nm: String;
        enforce domain left x: Element {
              name = nm,
              context = c1 : Context {}
        };
        enforce domain right y: Element {
              name = nm,
              context = c2 : Context {}
        };
        when {
              RootMapping(c1,c2) or
CompositeFlattening(c1,c2);
        }
}
```

Examples: Below we have applied this pattern to flatten the Block hierarchy from a Shapes model. The additional condition `not(RootBlockMapping(b1,b2))` is required to make sure that the Block b1 is not the RootBlock.

```
top relation BlockFlattening {
        checkonly domain left b1: Block {
                block = c1 : Block {} };
        enforce domain right b2: RootBlock {};
        when {
                not(RootBlockMapping(b1,b2));
   RootBlockMapping(c1,b2) or
BlockFlattening(c1,b2);
        }
}
```

Applicability: The flattening pattern can be used to remove hierarchical structure from a model.

6. Applying transformation patterns

Transformation specifications are made up of rule definitions (see Section 4.1). Each rule tackles a small part of the transformation problem. Transformation patterns can help to identify solutions to these partial transformation problems. In this section, we show how a complete transformation definition can be constructed and specified by combining rule definitions, which in turn are obtained by applying the rule patterns. The example illustrates how several different rule patterns are combined to provide a complete solution for a particular transformation problem. To demonstrate the viability of the approach the transformation is applied to statecharts, which is a well-known and frequently used formalism of UML. The **problem statement** is:

Given a hierarchical statechart, i.e., a statechart with composite states, produce a flat statechart without any hierarchy describing the same behaviour (Figure 4).

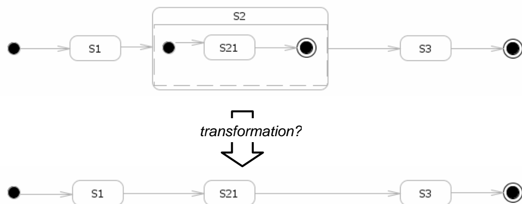


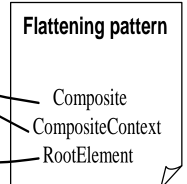
Figure 4. Statechart problem definition

We start the transformation specification with the declaration of the source and target domains. The source and target models are of the same type here, i.e., a statechart (see Figure 2 for the statechart metamodel).

```
Transformation
StatechartFlattening(left:StateChart,right:StateChart) {}
```

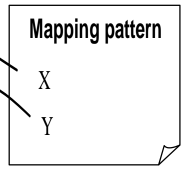
Obviously, the problem statement is a Flattening problem. Thus we first apply the Flattening pattern to define a rule for flattening composite states. Here the Composite is a CompositeState, the CompositeContext is a Container (which is another CompositeState or the StateMachine), and the RootElement is a StateMachine.

```
top relation CompositeFlattening {
    enforce domain left cs: CompositeState
    }; {
        container = c1 : Container {}
    enforce domain right sm: StateMachine {};
    when {
        StateMachineMapping(c1,sm)      or
          CompositeFlattening(c1,sm);}}
```



The above rule depends on a mapping between the root elements, i.e., the encompassing state machines. This relation is a simple instance of the Mapping pattern, in which a state machine on the left is related to a state machine on the right, such that their names are equal. As StateMachine is the root hierarchical concept no ContextMapping needs to be specified.

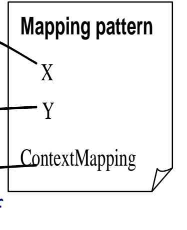
```
top relation StateMachineMapping {
    nm: String;
    enforce domain left sm1: StateMachine {
        name = nm};
    enforce domain right sm2: StateMachine {
        name = nm};}
```



The remaining elements of the state machine are also instances of the Mapping pattern. The rule for transforming SimpleStates, for example, is obtained by instantiating the Mapping pattern.

Also transitions between states from the source model are simply mapped onto transitions in the target model (as shown below), which is an instance of the Relationship Mapping pattern.

```
top relation SimpleStateMapping {
    nm: String;
    enforce domain left s1: SimpleState {
        container = c1 : Container {},
        name = nm};
    enforce domain right s2: SimpleState {
        container = c2 : Container {},
        name = nm};
    when     {StateMachineMapping(c1,c2)     or
          CompositeFlattening(c1,c2);}}
```



In the statechart metamodel, a Vertex is defined as a generalization of a State that can be used to distinguish between different types of states, e.g., Start State or Final State. It has been introduced to cope with a deficiency in the transformation execution engine to handle enumerations.

```
top relation TransitionMapping {
    nm: String; g: String;
    enforce domain left t1: Transition {
       container = c1 : Container {},
        source = ss1 : Vertex {},
        target = ts1 : Vertex {},
        name = nm,
        guard = g};
    enforce domain right t2: Transition  {
        container = c2 : Container {},
        source = ss2 : Vertex {},
        target = ts2 : Vertex {},
        name = nm,
        guard = g};
    when {
        StateMachineMapping(c1,c2)          or
        CompositeFlattening(c1,c2);
```

A
AContext
AS
AT
B
BContext
BS
BT

In order to obtain a semantically correct model, we additionally need to remove the initial and final states of all composite states. Moreover, we need to make sure that transitions that originally had a composite state as their target are now redirected to the target of the outgoing transition of the initial state of that composite state. And, conversely, that transitions that originally had a composite state as their source are now moved to the source of incoming transitions of the final state of that composite state. In principle we can do this by applying the node abstraction pattern, which takes a node and replaces it by a simpler structure. This pattern can be applied twice, first to remove the composite states and second to remove initial and final states. The next figure depicts the abstraction of pseudostates.

```
top relation InitialStateAbstraction {
    nm1, nm2: String;
    checkonly domain left ps: PseudoState {
        kind = PseudostateKind::pk_initial,
        container  =  c1  :  CompositeState  {
            incoming = inc : Transition {
              source = s1 : Vertex {},
              name = nm}},
        outgoing = out : Transition {
            target = t1 : Vertex {},
            name = nm2}};
    enforce domain right t: Transition {
        container = c2 : Container {},
        source = s2 : Vertex {},
        target = t2 : Vertex {},
        name = nm1 + nm2};
   when           {CompositeFlattening(c1,c2);
        VertexMapping(s1,s2);
```

Abstraction pattern
X

The next figure depicts the abstraction of the FinalState by instantiating the node abstraction pattern.

```
top relation FinalStateAbstraction {
    nm1, nm2: String;
    checkonly domain left fs: FinalState {
        container = c1 : CompositeState {
            outgoing = tr : Transition {
                target = t1 : Vertex {},
                name = nm1}},
        incoming = inc : Transition {
            source = s1 : Vertex {},
            name = nm2}};
    enforce domain right t: Transition {
        container = c2 : Container {},
        name = nm1 + nm2,
        source = s2 : Vertex {},
        target = t2 : Vertex {}};
    when         {CompositeFlattening(c1,c2);
```

Abstraction pattern
X

By combining all these transformation rules a transformation specification is obtained that is able to flatten the statechart.

7. Conclusions

Writing model-to-model transformations can be a tedious undertaking. In most model transformations there are certain underlying principles that can be used to facilitate the production of model transformations.

This paper has identified basic transformation patterns that frequently occur in model-to-model transformations such as the mapping pattern, the duality pattern, the refinement/abstraction pattern, the flattening pattern. These patterns have been described and specified in QVT Relations, resulting in a catalogue of basic transformation patterns. A simple Shape language has been introduced to illustrate most of the patterns. The catalogue of transformation patterns provided in this paper is a first attempt to categorize transformation principles in QVT Relations. This list is, however, not complete. A natural way to enrich this collection of pattern, would be to try to join our approach with similar initiatives in this area, such that the Modelware project ([1]). It remains however to investigate to what extent patterns proposed in [1] are implementable using the declarative QVT languages. Furthermore, we challenge the community to elaborate on this kind of work and extend the list of patterns.

Composition of model-to-model transformation should be guided, in our view, by the usability of the resulting transformation. In this respect we believe that it is not always meaningful to compose patterns. In practice, some particular compositions of patterns will occur more frequently than others. For example, the mapping pattern is often composed with many other patterns, but composition of node refinement with duality seems to make less sense in practical situations. Nevertheless, an analysis of pattern compositionality and parameterization makes the object of future work.

Acknowledgments

The work presented in this paper is part of the Freeband A-MUSE project (http://a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025.

References

[1] Allilaire, F., Bézivin, J., Olsen, G., Bailey, T., Bonet, S., Mantell, K., Vogel, R.: "D1.6-3 Identification of Transformation Patterns", FP6-IP 511731 MODELWARE, 04/09/06, http://www.modelware-ist.org/index.php?option=com_remository&Itemid=79&func=fileinfo&id=132 (1-2-2008).

[2] Alexander, C., Ishikawa, S., Silverstein, M.: "A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)", Oxford University Press, 1977.

[3] Brahe, S. and Bordbar, B.: "A Pattern-based Approach to Business Process Modeling and Implementation in Web Services", In Workshop Proceedings of the 4th International Conference on Service-Oriented Computing ICSOC 2006, Chicago, IL, USA, December 4-7, 2006, Lecture Notes in Computer Science, Vol. 4652/2007, pp. 166-177, ISBN 978-3-540-75491-6.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Element of Reusable Object-Oriented Software". Published by Addison-Wesley, 1995. ISBN 0201633612. 27th printing, November 2003.

[5] Judson, S.R., Carver, D.L., France, R.B.: "A Meta-Modelling Approach to Model Transformation". In: OOPSLA'03, p. 326-327, October 26-30, 2003, Anaheim, USA.

[6] Miller, J. and J. Mukerji (eds.): "MDA Guide Version 1.0.1", Object Management Group, June 2003.

[7] Object Management Group: "OMG Unified Modeling Language: Superstructure Version 2.1.1", 2003, http://www.omg.org/docs/formal/07-02-03.pdf (31-1-2008).

[8] Object Management Group: "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", Final Adopted Specification ptc/05-11-01, Nov. 2005, http://www.omg.org/docs/ptc/05-11-01.pdf (1-2-2008).

[9] Soley, R. and the OMG Staff Strategy Group: "Model Driven Architecture", Object Management Group White Paper, Draft 3.2, Nov. 2000.

[10] Wiki patterns catalogue: https://doc.telin.nl/dsweb/View/Wiki-90/HomePage.