Project Work

Anna-Maria Liebhoff

Reusable QVT patterns for state machine model transformations and their verification in VMTS

September 4, 2013
Project initiated by

Airbus Operations GmbH
Department ECYE22 - A350 / A380 Software
Lüneburger Schanze 30
21614 Buxtehude, Germany

Tutors

Alexander Wassew

and

Daniel Kliem

work by
Anna-Maria Liebhoff
born on 26th September, 1989 in Dresden
studying
Joint Master in Communication and Information Technologies
matriculation number: 43189
Hamburg University of Technology
Abstract

Model based software development requires several design steps. In each step the models change according to the level of abstraction. Model transformation helps to process the models without losing information. In this work, four transformation patterns are introduced. They contain typical procedures for converting models from one design level to another. For this project, a specialization to state machines was chosen because state machines play an important role for software development teams of Airbus. Since 2008 a standard model transformation language exists: Query/View/Transformation Language (QVT). It is used for representing the patterns. Additionally, a second modeling system is used. The Visual Modeling and Transformation System (VMTS) is based on graph-rewriting techniques and therefore a transformation can be validated. This fact is used for verifying the presented model transformation patterns.
Contents

List of Figures vii
List of Tables vii
List of Abbreviations ix

1 Introduction 1

2 QVT Transformation Language and VMTS Verification Framework 5
  2.1 QVT ................................................................. 5
  2.2 VMTS .............................................................. 7
  2.3 Conversion ...................................................... 9
  2.4 Software ......................................................... 9

3 Transformation Patterns for State Machines 11
  3.1 State Machine Metamodels ................................... 11
  3.2 The Mapping Pattern ......................................... 12
  3.3 The Refinement Pattern ...................................... 16
  3.4 The Node Abstraction Pattern ............................ 18
  3.5 The Flattening Pattern .................................... 19

4 Verification of Transformation Patterns with VMTS 23
  4.1 Deduction Rules ............................................... 23
  4.2 Assertion Sets ................................................ 26
  4.3 Verified Properties ........................................ 27

5 Case Study – Use of Transformation Patterns 31
  5.1 Transformation ............................................... 31
  5.2 Discussion ..................................................... 36

6 Conclusion 39
  6.1 Results ......................................................... 39
  6.2 Future Work ................................................ 40

Bibliography I
List of Figures

1.1 Transformation Concept Overview ........................................... 2
2.1 Common Graph Patterns ..................................................... 7
2.2 A Rewriting Rule .............................................................. 7
3.1 Metamodel of State Machines in SysML .................................. 12
3.2 Metamodel of State Machine in UML ..................................... 12
3.3 QVT Mapping Relations as VMTS Rewriting-Rules .................. 15
3.4 VMTS Control Flow Graph of QVT Mapping Transformation .... 15
3.5 VMTS Rewriting Rule of Transition Refinement Pattern .......... 18
3.6 VMTS Rewriting Rules of Refinement Patterns ...................... 18
3.7 VMTS Rewriting Rule of State Element Abstraction Pattern .... 19
3.8 VMTS Rewriting Rules of Flattening Pattern ......................... 22
3.9 Control Flow Graph for Flattening Transformation ................ 22
4.1 Graph Patterns of Rewriting Rules ....................................... 27
5.1 System State Machine in Classic Lighting Mode ..................... 37
5.2 Software State Machine in Classic Lighting Mode ................. 37

List of Tables

2.1 Syntax of a Formula of an Assertion ...................................... 8
4.1 Initial Assertion Sets for the Transformation Patterns ............ 26
4.2 Extended Assertion Set for the Transformation Patterns .......... 28
4.3 Assertions of Verified Properties of the Transformation Patterns . 29
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Assertion Description Language</td>
</tr>
<tr>
<td>ATL</td>
<td>Atlas Transformation Language</td>
</tr>
<tr>
<td>CF</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>DPO</td>
<td>Double Push-Out</td>
</tr>
<tr>
<td>LHS</td>
<td>Left-Hand-Side</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation Language</td>
</tr>
<tr>
<td>RFP</td>
<td>Request for Proposal</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-Hand-Side</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VMTS</td>
<td>Visual Modeling and Transformation System</td>
</tr>
</tbody>
</table>
1 Introduction

Model transformation is an important concept in model based engineering. Many paradigms are connected to this topic and there are different solutions and suggestions for the same problems. The chain of concepts used in this work is introduced first.

Transformation and Transformation Languages An important feature of the Model Driven Architecture (MDA) concept of the Object Management Group (OMG) is model transformation. Models at different levels of abstraction and also in different modeling languages can be transformed into each other. In 2002 the OMG set up a Request for Proposal (RFP) to get a standard transformation language: Query/View/Transformation Language (QVT). Today several implementations of model transformation tools are on the market. Most of them use their own transformation language, which is only related to QVT. An example of this is the open source Eclipse plug-in Atlas Transformation Language (ATL). [Noi09]

Nevertheless, the setup is the same in all tools and languages. Each model that should be transformed into another one is related to a so called metamodel. A metamodel is usually a UML class diagram representing a modeling language. In a transformation script, the mapping of the different objects of the source-(meta)model to the target metamodel is described. The target model might have the same metamodel (endogenous transformation) or a different one. The transformation engine takes the metamodels, the script, and a specific model as input and creates a specific output model. In figure 1.1 this and other concepts of this thesis are presented.

System and Software Modeling in Industry This work is based on a project with Airbus Operations GmbH. In the different steps of the software development process, different models are created. The software model (Platform Specific Model (PSM)) is, until now, created manually from the system model (Platform Independent Model (PIM)). Especially because there are different teams working on the different stages, it is not assured that the different models represent the same system in every detail. Airbus developers wish to have means to verify if their manual transformation is correct. [Kli13]

Science knows about this problem and also draws high attention to the PIM to PSM transformation. [ISH08]

Metamodels of State Machines As described in the first paragraph, metamodels are needed for the transformation. They present how models are constructed, meaning they formally express a modeling language. This work concentrates on state machines. Two different metamodels are used. System designers create their models with the modeling language SysML, so the metamodel constitutes a state machine in SysML. The software developers, who are creating code out of the designer’s model, use UML for modeling. That is why the second metamodel, constituting a state machine in UML, is similar but not the same as the first one.

SysML is an extended subset of UML. It is developed for systems engineering and defines, e.g., additional diagram types.[Wei08]
In order to show the transformation between different metamodels, without rising the complexity, the same metamodel structure with different names is used.

**Verification of Model Transformations** One distinguishes between two types of verification of model transformations. *Online* verification compares the input and output models directly. Model checking is applied in combination with certain correctness properties that have to be verified. *Offline* verification, on the other hand, is related to the transformation script. It needs higher effort, but every transformation using this script with any input model is automatically proved. Input and output models are not taken into account. In contrast, *online* verification must be done after every single transformation process on the input and output model. This can become difficult for larger systems due to the state explosion problem caused by the model checking techniques [Var02]. Therefore, the *offline* approach is used in this work. Since graph-rewriting methods are very well developed, they are used for verification.

**Transformation Patterns** Design patterns are widely spread in software development. They are solutions to frequently occurring problems. Often, developers have similar tasks and not everyone should invent a new solution, but use approved standards instead. For model transformation scripts, there are also some ideas for patterns. In the paper [ISH08] the authors introduce a library for transformation patterns and suggest some that are based on their work with QVT. [Nov] Unfortunately, this library is not updated and the number of collected patterns did not grow.

**Model Transformation Tools** There are some transformation tools on the market. The above mentioned ATL Eclipse plug in is one example. For QVT there is smartQVT and mediniQVT. Most of the tools are, unfortunately, not well maintained or commercial.
software. In the Paper [ALL10], a verification method is described that uses the Visual Modeling and Transformation System (VMTS). It is both method and tool and addresses functional properties. It is based on graph transformation techniques, which are also applied here. Another verification framework is called VIATRA [Var02] and aims to verify syntactic and semantic correctness. For this work VMTS is chosen because for Airbus developers it is the most important to have functional properties proved, because they want to preserve the system behavior during a transformation.

From QVT to VMTS  VMTS, used for the verification of the patterns, has its own transformation language. Though, it has a plug-in to convert QVT scripts to the graph-rewriting approach. [LLVC06] The verb to convert here could also be to transform. For clarity, in this work conversion and convert are only used for the language transformation from QVT to VMTS language and transformation and transform for model transformations.

Aim of this work  In short, this work aims to offline verify four transformation patterns: the Mapping Pattern, Refinement Pattern, Node Abstraction Pattern and Flattening Pattern. Airbus developers can use these for checking their own transformations. They can combine the patterns in a way that they get the desired software model from the system model. By this, the transformation is verified and documented in the same time. The formal verification deals with functional properties of the models.

In chapter 2, QVT, VMTS and their conversion is presented. After that, in chapter 3, transformation patterns for state machines are introduced and their correspondence in VMTS is given. Chapter 4 is about the verification of the transformation patterns, while in chapter 5 a sample transformation of an Airbus system is described. This case study shows how the patterns can be connected. Finally, in chapter 6, the results of this work are summarized and an outlook to future work is given.
2 QVT Transformation Language and VMTS Verification Framework

Model transformations can be defined in various ways and languages. The OMG specifies QVT [Omg08], which might become the standard transformation language. This opinion is shared by the authors of [LLVC06]. Accordingly, the patterns in this work are described in QVT.

However, graph transformations have a long research history and a reliable formal background is given [LLVC06]. The idea to use graph transformation for model transformation is realized in VMTS, which is a framework and tool to specify models and execute model transformations. It also helps to verify the transformations with its graph-rewriting based methods. [OTEI10]

This chapter gives an introduction to both QVT and VMTS. It also shows how to convert QVT into VMTS.

2.1 QVT

The Query/View/Transformation Language-principle can be described as follows. Queries are related to a source model and figure out candidates for the transformation. They can be sub-models or the source model itself. Views are, accordingly, related to a target model and describe how the result of a transformation should look. The Transformation itself is the process that transfers the concrete source model, determined by a query, into the target model. [Nol09]

QVT actually is a set of three languages, which are Operational Mappings, Core Language and Relations Language. Operational Mappings is an imperative language that allows transformations in one direction. Performed changes cannot be reconstructed in the source model. Core and Relations Language are descriptive languages that relate source and target model (Query and View) symmetrically. The Relations Language also provides rule descriptions using the Object Constraint Language (OCL) and is completely transformable to the Core Language. The Core Language is the most basic transformation language but can describe any Model-to-Model (M2M) transformation. Any transformation language in accordance with the MDA principle can be mapped to the Core Language. [LLVC06] [Nol09]

The Relations Language is used in this work, because of its powerful expressiveness and strong connection to source and target model. It also provides bi-directional transformation.

General Structure of a QVT Relations Language Script

The following listing shows the syntactical composition of a transformation script in QVT Relations Language. The transformation patterns, which are introduced later, are expressed in this language.
A transformation relates metamodels with each other. They are the arguments of the transformation declaration.

**transformation** Hello World ( source: UML , target: UML ) { ... }

The transformation consists of one or more relations, which specify rules for the model elements. The keyword **top** marks **top-level** relations, which are implicitly executed if the model element is found in the source model. **Non-top-level** relations have no further marking and are only executed when they are called. In each transformation there must be at least one **top relation**.

**top relation** SourceToTarget { ... }

The rules of a relation are expressed in domains. A domain pattern represents an element of one of the related models. The word **pattern** in this context has nothing to do with the transformation patterns. Mostly, in one relation one domain per related model is specified. The keyword **checkonly** tests the validity of a domain pattern while **enforce** creates validity. The following listing transforms an empty model into a model that contains a package named **HelloWorld**.

**transformation** Hello World ( source: UML , target: UML ){ 
**top relation** SourceToTarget 
  {  
    **checkonly domain** source srcPckg: Package {};  
    **enforce domain** source srcPckg: Package  
    {  
      name= 'Hello_World'  
    };  
  }  
}

**Relations** can contain variables that are used in several domains. Additionally, **when** and **where** predicates can be defined. The **when** clause is processed first. Here, other relations might be called. It is a pre-condition. The **where** condition is an invariant. The clause is executed always when a variable is modified.
The constructs **key**, **primitive domain** and **query** are optional and not relevant for this project work. A further description is available in [Omg08] or [Nol09].

2.2 VMTS

While QVT is mainly used for defining model transformations, other implementations concentrate on the execution. VMTS is one of them. For a transformation a Control Flow Graph (CF) is defined in which the order of graph-rewriting rules is specified. So called assertions are added to specific points in the CF. These three basic concepts are forming a transformation in VMTS. Since it is working with graph logic, the notation is also graphical. [ALL10]

*Graph patterns* are used to express reused graph elements.

In contrast to the QVT Relations Language, graph rewriting systems do not work bidirectional.

**Graph Patterns** The word *pattern* is used in several contexts in this work. Graph patterns are small parts of graphs or models. They are matched as subgraphs in larger graphs or models. Figure 2.1 shows three typical graph patterns. More graph patterns, which are used in this work, are presented in figure 4.1.

![Figure 2.1: Common Graph Patterns](image)

**Rewriting Rules** A rewriting rule expresses how a part of a model shall be transformed. It is composed of a small graph, called Left-Hand-Side (LHS), and an corresponding Right-Hand-Side (RHS) graph. The LHS is supposed to be found within a source model and then be replaced by the RHS. By this the target model is formed. LHS and RHS are instances of elements of the metamodels. Figure 2.2 presents a typical rewriting rule.

![Figure 2.2: A Rewriting Rule](image)

On the left, a graph pattern is displayed (LHS). If this is found in the source model, the sub-model will be changed to the graph pattern on the right of the rewriting rule (RHS). The target model is the outcome.

**Control Flow Graph** The CF in VMTS contains the control structure of a transformation. It has a start node to which the source model is assigned and an end node from where the target model can be taken. All the other nodes contain each one rewriting rule. The graph can contain loops and branches that are depended on constraints.
During execution, a model is passed through the graph and modified in each step. The graph is based on a Unified Modeling Language (UML) activity diagram. Like typical dataflow, each target model of a rule in node \( i \) is source model of the rule in node \( i + 1 \).

**Assertion Set**  Assertions are logic expressions that define constraints at specific points in the execution of a transformation. They refer to properties of the model at a certain point of the CF and to properties of the rewriting rules. For example, no element of type \( P_x \) is contained in the current model, the rule terminates or a rule creates elements of type \( P_x \). Here, \( P_x \) is a graph pattern. For a transformation there exists an initial assertion set. It describes the properties of a model in the beginning of the transformation. With the help of deduction rules, more assertions, further in the transformation, can be derived and proved. This can be used in order to verify certain additionally desired properties of a transformation. More about this is written in chapter 4.1.

Assertions are expressed in Assertion Description Language (ADL). An ADL statement consists of two parts: a prefix (\( \text{pre}(s) \), \( \text{post}_\text{succ}(s) \), \( \text{post}_\text{fail}(s) \), \( \text{at}_\text{succ}(s) \), \( \text{at}_\text{fail}(s) \), before\( (s) \) and after\( (s) \)) and a dynamic or static Formula. While the prefix describes the position in the CF relative to a step \( s \), the Formula describes a property of the model at that point of execution. The at prefix is for dynamic, all others are for static Formulas.

An example for an assertion is: after\( (\text{rule1}) \) None \( P_x \). It expresses that after executing \( \text{rule1} \) there are no subgraphs of graph pattern \( P_x \) contained in the model, if the assertion holds. Table 2.1, taken from [ALL10], presents the syntax of the Formulas.

<table>
<thead>
<tr>
<th><strong>STATIC FORMULAS IN ADL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name:</strong> None</td>
</tr>
<tr>
<td><strong>Syntax:</strong> None ( P )</td>
</tr>
<tr>
<td><strong>Semantic:</strong> No instances of pattern ( P ) exist in the model.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NAME</strong></th>
<th><strong>SYNTAX</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Exists</td>
<td>( P_1 \rightarrow_m P_2 )</td>
</tr>
<tr>
<td>Semantic</td>
<td>( m ) is a model pattern graph morphism from ( P_1 ) to ( P_2 ). It is true that for each instance ( I_1 ) of pattern ( P_1 ) an instance ( I_2 ) of pattern ( P_2 ) can be found, such that the following conditions hold: ( (i) ) if ( m \in N_{P_1} \land m(n) = n' \in N_{P_2} \Rightarrow n(I_1) = n'(I_2) ) ( (ii) ) if ( e \in E_{P_1} \land m(e) \notin E_{P_2} \Rightarrow e(I_1) = e'(I_2) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DYNAMIC FORMULAS IN ADL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NAME:</strong> ForOne ( P_1 \rightarrow_m P_2 )</td>
</tr>
<tr>
<td><strong>Syntax:</strong> ForOne ( P_1 \rightarrow_m P_2 )</td>
</tr>
<tr>
<td><strong>Semantic:</strong> ( m ) is a model pattern graph morphism from ( P_1 ) to ( P_2 ). An instance ( I_1 ) of pattern ( P_1 ) is matched from the model and will be replaced with an instance ( I_2 ) of pattern ( P_2 ) such that: ( (i) n \in N_{P_1} \land m(n) \notin N_{P_2} \Rightarrow n(I_1) ) will be deleted ( (ii) e \in E_{P_1} \land m(e) \notin E_{P_2} \Rightarrow e(I_1) ) will be deleted ( (iii) n \notin N_{P_1} \land m(n) \in N_{P_2} \Rightarrow n(I_1) ) will be a newly created node in the model ( (iv) e \notin E_{P_1} \land m(e) \in E_{P_2} \Rightarrow e(I_1) ) will be a newly created edge in the model.</td>
</tr>
</tbody>
</table>

| **NAME:** ForEach \( P_1 \rightarrow_m P_2 \) |
| **Syntax:**ForEach \( P_1 \rightarrow_m P_2 \) |
| **Semantic:** A for each means that the ForEach \( P_1 \rightarrow P_2 \) modification is applied repeatedly until it cannot be applied any more. |

| **NAME:** Termination |
| **Syntax:** Termination |
| **Semantic:** The current step terminates. |

Table 2.1: Syntax of a Formula of an Assertion [ALL10]
2.3 Conversion

The conversion of QVT to VMTS is essential for this work. Model transformation patterns are developed in a standard language, while they are verified in an approved verification system. In the paper [LLVC06] is presented how QVT constructs can be converted to VMTS constructs. The authors also introduce a plug-in for the VMTS software, based on this research. Unfortunately, it emerged that understanding and running the plug-in exceeds the frame of the project. For this reason the conversion is done manually, according to the related work about the plug-in. The manual conversion of the transformation script is feasible because the QVT patterns are sufficiently simple, e.g. containing only one checked and one enforced domain per relation. Furthermore, this project does not focus on the QVT-VMTS conversion because VMTS is only the chosen verification framework and does not influence the transformation itself.

A common principle of both systems are the metamodels. They remain unchanged in the conversion. Similarly source models, instances of metamodels, will be the same in both systems.

In contrast, the change of a sub-model is expressed quite differently. A pattern described in QVT checked domain is transformed to an enforced domain. The patterns of both (or more) domains can be converted to LHS and RHS of a rewriting rule. The properties of the checked domain additionally become assertions located before the appearance of that rule in the CF.

The control is the second, even bigger, difference. In QVT, when and where clauses of relations define what happens before and after the realization of a relation. Top relations are always executed. Non-top relations are only realized if they appear in a when or where clause of another relation. Respecting these hierarchical structures, a CF can be constructed straight-forward.

Each relation of a transformation becomes a rewriting rule, brought in an execution order (CF) by its when and where clauses.

2.4 Software

The VMTS software can be downloaded from the homepage of University of Budapest [cTEB10]. When this project started, the files could be received, but when it progressed, the download of the verification plug-in was disabled.

A VMTS project consists of many models. The metamodels, each rewriting rule, the CF and the source model have to be graphically created when a transformation is set up. A user manual is not available and will not be provided with this work. The reason is, that it cannot be recommended to use the tool outside this work in daily routine of Airbus because of missing support from the developers’ part.

Still, this system was chosen because it has many good features like the QVT processor, the verification plug-in and its formal background and documentation ([Asz12]). They are just not ready for productive use. Additionally, authors of many papers, more than 150 [cTEB10], recommend VMTS. Furthermore, there are no better alternatives to this system that verifies functional properties offline.
3 Transformation Patterns for State Machines

In software development, design patterns give general solutions to problem statements. In this chapter, four of them are motivated and defined: the Mapping Pattern, Refinement Pattern, Node Abstraction Pattern and Flattening Pattern. In this work, the framework is specified for state machines, in contrast to the related work [ISH08] on which these patterns are based. In [ISH08] five transformation patterns are defined generally without restriction to specific metamodels. The fifth pattern, which is not introduced in this work, is the Duality Pattern. It relates diverse model types with each other and does not add value to the main topic of this work. Therefore, it is not discussed here. Before introducing the patterns, the state machine metamodels are described.

3.1 State Machine Metamodels

A metamodel is a model, usually expressed in a UML class diagram, which describes the syntax of a modeling language. It shows which elements can appear in that language and how they are related.

The OMG defines that anything which is described in a modeled language can be transformed. Also the metamodeling language, UML class diagram, can be expressed through a model. This is then called metametamodel. It can be seen that metamodeling takes place in in several layers. Each model can be a metamodel of another and each model is an instance of its metamodel.

In this project a simple model of a state machine, also called state chart, is used. Transformation from SysML models to UML models are made. The figures 3.1 and 3.2 present both metamodels. They are created on the basis of [MS]. There is only a difference in naming of the classes, because one aim of this project is to show how to implement and express a transformation between different model kinds. Nevertheless, it is a feasible simplification because SysML is a subset of UML. State machines are contained in both specifications and only have different identifiers. A few features of SysML are not important for this project. [Wei08]

In [Nol09] a transformation between more diverse modeling languages, state charts and relational database models, is described. It is a typical example, referenced in several papers. [LLVC06] [Omg08]

States (StateElements) are connected through transitions. In this simple representation of state charts there are four special cases of a simple state. The start (StartState) and end state (EndState) identify the beginning and end of the state machine execution. A super-state (StateItem) can contain a whole sub state machine. It is also the most outer frame. If a state chart is transformed the basic instance StateItem, containing the rest of a specific model, is mapped first. Furthermore, there is the history state (HistoryIndicator) which is a special construct of state machines. It saves the last state when a super-state is left. If a transition is going to the history state, the next state will be the saved one.
3.2 The Mapping Pattern

The first pattern that is introduced is the most basic application of transformations. It is the direct mapping between two elements. Especially, if there are two different modeling languages, it is obvious that a rule is needed to determine which element of the source language corresponds to which one in the target language. But also for all the other transformation patterns, this pattern is needed.

**General QVT Pattern**

The following QVT code describes in general the mapping of an element. For each element type of a model one relation has to be defined. These element types are the ones defined in the metamodel(s). The type of an element is expressed in the QVT code by \( x: \text{type} \{ \} \). In the brackets, further definitions of the type can be stated. When parts or properties of the element also have to be related, this is defined in the \textbf{when}-clause. An example of this is the mapping of a transition. The relation describes how it is represented in the source and target model but also its start and end nodes have to be mapped to the respective nodes in the other model.

The elements have contexts. Elements in the same hierarchical level belong to the same context. That is why a relation usually has a \textit{ContextMapping()}\ in its \textbf{when}-clause. Even in a flat model without, e.g., super-states in state machines, all instances belong to the
con text \textit{state machine}. The \textit{state machine} is an instance of \textit{stateItem} and the most outer frame.

The calls in the \textit{when-} (and \textit{where-}) clauses are relations which are defined somewhere else in the transformation script.

Only elements of \textbf{enforced domains} are created, so, every element of the target model has to appear in at least one relation with \textbf{enforced domain}. On the other hand, all elements, and only these ones, which correspond to the \textbf{checkonly domains}, are transformed. The \textbf{domains} in a relation describe patterns. All elements of the input model that match these types will be transformed to the defined target \textbf{domain} pattern. \cite{Noi09}

Name clashing is not an issue because all identifiers belong to their \textbf{domain}. If a name shall be adopted, it is explicitly defined by a shared variable, e.g. in the code: \textit{nm: String}.

\begin{verbatim}
  top relation elementMapping {
    nm: String;
    checkonly domain source element a: elaType {
      context = c1 : AContext {}
      name = nm
      otherProp = a : A {}
    };
    enforce domain target element b: elbType {
      context = c2 : BContext {}
      name = nm
      otherProp = b : B {}
    };
    when {
      ContextMapping(c1, c2);
      otherPropMapping(a, b);
    }
  }
\end{verbatim}

\cite{ISH08}

\textbf{Specific QVT Pattern}

Now the pattern is given that facilitates the mappings of all elements of the state machines. In order to see how to use the pattern and also how it works, the string \textit{_trafo} is added to the name of an element.

\begin{verbatim}
  transformation directMapping (source: sysml, target: uml) {
    relation StateItemMapping {
      nm: String;
      checkonly domain source s1: sStateItem {
        name = nm
      };
      enforce domain target s2: uStateItem {
        name = nm + '_trafo'
      };
      --no context and when statement because it is the most general concept
    }
    top relation StateElementMapping {
      nm: String;
      checkonly domain source s1: sStateElement {
        container = c1 : sStateItem {}
        name = nm
      };
      enforce domain target s2: uStateElement {
        container = c2 : uStateItem {}
      }
  }
\end{verbatim}

13
name = nm + ' _trafo'
}
when {
    StateItemMapping(c1, c2);
}

relation HistoryMapping {
    nm: String;
    checkonly domain source a: sHistoryIndicator {
        container = c1 : sStateItem {},
        name = nm
    };
    enforce domain target b: uHistoryIndicator {
        container = c2 : uStateItem {},
        name = nm + ' _trafo'
    };
    when {
        StateItemMapping(c1, c2);
    }
}

top relation sStateMachine {
    nm: String;
    checkonly domain source a: sStartState {
        container = c1 : sStateItem {},
        name = nm
    };
    enforce domain target b: uStartState {
        container = c2 : uStateItem {},
        name = nm + ' _trafo'
    };
    when {
        StateItemMapping(c1, c2);
    }
}

top relation eStateMachine {
    nm: String;
    checkonly domain source a: sEndState {
        container = c1 : sStateItem {},
        name = nm
    };
    enforce domain target b: uEndState {
        container = c2 : uStateItem {},
        name = nm + ' _trafo'
    };
    when {
        StateItemMapping(c1, c2);
    }
}

top relation TransitionMapping {
    nm: String;
    checkonly domain source a: sTransition {
        container = c1 : sStateItem {},
        name = nm,
        source = as : sStateElement {},
        target = at : sStateElement {}
    };
    enforce domain target b: uTransition {

container = c2 : uStateItem {},
name = mn + '_trafo',
source = bs : uStateElement {},
target = bt : uStateElement {}
};
when {
    StateItemMapping (c1 , c2);
    StateElementMapping (as , bs);
    StateElementMapping (at , bt);
}

If one element is not included in the model to be transformed or shall not appear in the target model, the relation does not need to appear in the transformation script.

Pattern in VMTS

Graph-rewriting based model transformations can be verified formally, which is not the case for QVT transformations. For that reason, the pattern is now expressed in VMTS. The control flow graph can be derived from the **when**- and **where** statements of a relation. [LLVC06]

The order of processing the **rules** is given in the **CF**. Each relation referenced in a relation’s **when**-statement has to be processed before the referencing one. Likewise, a relation called in the **where**-statement has to be processed afterwards. Each relation in QVT corresponds to one (rewriting-)rule in VMTS. (See also chapter 2.3)

In figure 3.3 the mapping rules are presented.

![General Element Mapping Rule](#) ![Transition Mapping Rule](#)

Figure 3.3: QVT Mapping Relations as VMTS Rewriting-Rules

The relations **StateItemMapping**, **StateElementMapping**, **HistoryMapping**, **sStateMapping** and **eStateMapping** all look like the general rule on the left. The relation **TransitionMapping** belongs to the rule on the right. According to the QVT script given in the previous section, the control flow graph for VMTS transformation can be seen in figure 3.4.

![Figure 3.4: VMTS Control Flow Graph of QVT Mapping Transformation](#)
3.3 The Refinement Pattern

In the development phase a model needs to be specified in more detail several times. The refinement pattern facilitates this process. A node can be split up into a more detailed system, as well as an edge can be. One distinguishes between relation refinement and node refinement. If a system more complicated than only one element needs to be refined, the pattern has to be applied several times.

General QVT Pattern

The following QVT code describes in general the refinement of an element. Each element that shall be refined has incoming connections, e.g., incoming edges or a source node, and outgoing connections. These connections are also the frame connections of the refined subsystem in the target model. They are usually of a different element type than the element to be refined. For example, a state of a state machine shall be refined, but the elements with which it is connected to the surrounding model are transitions. They are transformed with the mapping pattern.

top relation elementRefinement {
    nm : String;
    -- the source element --
    check only domain source element1 : a1Type {
        name = nm,
        context = c1 : AContext {},
        incon = a2_in : a2Type {},
        outcon = a2_out : a2Type {}
        --more connections possible --
        --to be considered also in enforced domains--
    };
    -- an intermediate element--
    enforce domain target im_element {
        context = c2 : BContext {}
    };
    -- more intermediate elements possible --
    enforce domain target elementb11 : b1Type {
        incon = b2_in : b2Type {},
        name = b2_in.name + '_con_to_' + im_element.name,
        context = c2,
        outcon = im_element
    };
    enforce domain target elementb12 : b1Type {
        outcon = b2_out : b2Type {},
        name = im_element.name + '_con_to_' + b2_out.name,
        context = c2,
        incon = im_element
    };
    when {
        ContextMapping (c1, c2);
        ElementMapping (a2_in, b2_in);
        ElementMapping (a2_out, b2_out);
    }
}

This general pattern puts an element of another kind in the middle of two elements of the type that shall be refined. The following example is related to state machines and should be more clear.
It is also possible to create an even more detailed system with a parallel state. In this case the number of enforced domains has to be doubled. Additionally, the names have to be given differently in order to not create elements with same identifier.

**Example of specific Pattern**

For the refinement pattern for state machines the relation refinement is given here. The node refinement can be derived from the general pattern in the same way. The user of the pattern might change it slightly, for example to detail an element with another structure. This is usual for design patterns. The following transformation takes a transition and creates a transition-state-transition sub-system.

```
transformation Refinement (source: sysml, target: uml) {

top relation TransitionRefinement {

  nm : String;
  — the source element —
  checkonly domain source a : sTransition {
    name = nm,
    container = c1 : sStateItem {}
    source = as : sStateElement {}
    target = at : sStateElement {}
  }
  — an intermediate state —
  enforce domain target im_state {
    container = c2 : uStateItem {}
  }
  enforce domain target b1 : uTransition {
    source = b1s : uStateElement {}
    name = b1s.name + '_to_' + im_state.name
    container = c2
    target = im_state
  }
  enforce domain target b2 : uTransition {
    target = b2t : uStateElement {}
    name = im_state.name + '_to_' + b2t.name
    context = c2
    source = im_state
  }
  when {
    StateItemMapping (c1, c2);
    StateElementMapping (as, b1s);
    StateElementMapping (at, b2t);
  }
  — more relations —
}
```

**Pattern in VMTS**

The **VMTS** correspondence for the refinement pattern is comparatively simple because it has only one rewriting rule to be executed. In case of a more complex system to be refined, several rules are concatenated in the **CF**. Figure 3.5 presents the specific pattern for state machines given above while figure 3.6 shows more possible refinements that can be derived from the general pattern.
3.4 The Node Abstraction Pattern

In some transformations a node has to be removed from a model in order to make it, for example, conform to other-level-rules. The node abstraction pattern is applicable for all kinds of models that are based on graph kind models with nodes and edges. \cite{ISH08} A relational database model is for instance not capable of such a transformation.

**Pattern in QVT**

For this pattern there is no distinction between the general and the specific-for-state machine pattern because there is only a difference in naming. States are nodes and transitions are edges. The following QVT pattern for state machines removes a `stateElement`.

```qvt
stateAbstraction {
  checkonly domain source a : sStateElement {
    inTransition = t_in : sTransition {
      name = na_in : String,
      source = ss1 : sStateElement }
  },
  outTransition = t_out : sTransition {
    name = na_out,
    target = tt1 : sStateElement }
};

enforce domain target b : uTransition {
  name = na_in + na_out,
  source = ss2 : uStateElement }
```
The node abstraction pattern in VMTS is the reversed refinement pattern that is given in the example above and the CF is similarly simple. Not every refinement is the opposite of node abstraction. Figure 3.7 presents the rewriting rule for the node abstraction pattern.

![Node Abstraction Diagram](image)

**Figure 3.7: VMTS Rewriting Rule of State Element Abstraction Pattern**

### 3.5 The Flattening Pattern

The following pattern is used to remove hierarchical structures. In this case, super-states can be removed by mapping them to the higher level container until the root element, which is the model itself. The flattening pattern does only remove a container element. The redirection of incoming and outgoing edges must be done additionally.

**General QVT Pattern**

Even though the flattening patterns appears to be the most complicated, it has the fewest lines of code. Still, it removes the whole hierarchy due to its recursive structure:

```plaintext
top relation CompositeFlattening {
    checkonly domain source a : Composite {
        context = c1 : CompositeContext {}
    };
    enforce domain target b : RootElement{};
    when {
        RootMapping(c1,b) or
        CompositeFlattening(c1,b);
    }
}
```

The breaking criterion of the recursion is the root mapping. If the parent component is the most outer frame then the elements already belong to the lowest level. All elements are mapped to the same context to which their parent context is mapped to.

**Specific QVT Pattern**

In state machines each super-state contains a complete state machine with start and end state. Incoming transitions of a removed super-state have to be redirected to its start state and the outgoing transitions, accordingly, have to originate in the end state.
However, in order to receive a valid state machine, these start and end states, in the middle of normal states, have to be removed via the node abstraction pattern. Therefore, the incoming transitions are directly redirected to the first successive state of the former start state, outgoing transitions respectively. [ISH08]

All other states and transitions are transformed with the mapping pattern. The root element of our metamodels is a StateItem. It is the most general element and is, as root, called like the model itself: state machine.

transformation Flattening (source : sysml, target : uml) {

top relation CompositeFlattening {
  checkonly domain source superstate : sStateItem {
    container = cl : sStateItem {}
  };
  enforce domain target statemachine : uStateItem {};
  when {
    not StateMachineMapping(superstate, statemachine);
    StateMachineMapping(cl, statemachine) or
    CompositeFlattening(cl, statemachine);
  }
}

top relation StateMachineMapping {
  nm : String;
  checkonly domain source statemachine1 : sStateItem {
    name = nm
  };
  enforce domain right statemachine2 : uStateItem {
    name = nm
  };
}

top relation StateElementMapping {
  see Mapping Pattern
  when {
    StateMachineMapping(cl, c2) or
    CompositeFlattening(cl, c2);
  }
}

top relation TransitionMapping {
  see Mapping Pattern
  when {
    StateMachineMapping(cl, c2) or
    CompositeFlattening(cl, c2);
  }
}

top relation InitialStateAbstraction {
  nm1, nm2 : String;
  checkonly domain a : sStartState {
    container = cl : sStateItem {
      right = incoming : sTransition {
        source = s1 : sStateElement {},
        name = nm1
      }
    }
  }
}
This pattern is already a demonstration of how all the patterns can work together and transform more complex models.

**Pattern in VMTS**

In VMTS the flattening pattern is a bit more complex than the other patterns. In [ALL10] the flattening transformation is presented for business process models. Here, this approach is adapted to state machines. The transformation consists of three rules. The first one redirects the incoming and outgoing transitions of a super-state. The second rule removes all compositions to a super-state from each state and makes it part of the first hierarchical level. Removing of all super-states is done in the last rule. All rules are applied exhaustively.
meaning as often as instances of the pattern can be matched to the input model. In figure 3.8 these rewriting rules are presented.

Figure 3.8: VMTS Rewriting Rules of Flattening Pattern

The execution of the rules is sequential, concluding in the CF shown in figure 3.9.

Figure 3.9: Control Flow Graph for Flattening Transformation
4 Verification of Transformation Patterns with VMTS

In this chapter, the model transformation patterns are verified. This work aims to verify functional properties. Every rewriting rule is checked individually. Since offline verification is done, it has to be assured that the result holds for any input and output model pair.

With the introduction of a rewriting rule a basic assertion set is set up. It describes the properties of the rule. Additionally, some constraints on the input model can be defined. They extend the basic assertion set. Then, the functional properties that shall be verified are also expressed in ADL. In order to verify the properties they are derived from the basic assertion set with the help of deduction rules.[ALL10]

The VMTS verification plug-in is able to process these steps. According to the developer the input for the plug-in is generated automatically from a transformation defined in the VMTS main program.[Asz12] Unfortunately the according file cannot be found in the data structure and no support is given by the responsible person. Consequently, the following sections describe theoretically the initial assertion sets, the properties that are verified and the deduction rules. Which concrete properties of each pattern are proved is described in section 4.3.

4.1 Deduction Rules

Mathematical Background

Before introducing the deduction rules, the Double Push-Out (DPO) approach is presented. It is used for graph transformation and helps to formally define model transformations.

\[(L \leftarrow K \rightarrow R)\]

is a so called production or rewriting rule. L, K and R are finite graphs. L is the LHS, R the RHS and K the interface graph. l and r are graph morphisms. This diagram presents this concept together with the DPO concept:

\[
\begin{array}{c}
L \leftarrow l \rightarrow K \rightarrow r \rightarrow R \\
G \leftarrow f \rightarrow D \rightarrow g \rightarrow H \\
\end{array}
\]

(1) and (2) are push-outs. A rewriting rule is applicable if the push-outs exist. Match m expresses that an isomorphic occurrence of the LHS is found in the input graph G. The graph transformation can be executed. H is the output graph with the replaced LHS-subgraph by the RHS. Match m is also called injective match. This technique is used in most model graph transformation engines.[ALL10]

An assertion \textbf{ForOne} \(L \xrightarrow{p} R\) describes the above mentioned transformation. The graphs G and H are models. The morphisms l and r related to the interface K in the DPO diagram can be computed from morphism \(p : L \rightarrow R\).[ALL10]
Deduction Rules

In the following, five deduction rules are introduced. They are used to propagate None and Any formulas through the CF. For the input model, some constraints are set. These constraints are expressed as formulas that are true before the first transformation rule. Furthermore, assertions describing the rewriting rules are defined. With the help of the deduction rules it can be stated which formulas are true after the last rewriting rule.

All following deduction rules have premises in common. They are stated here to not be repeated in the description of each rule:

- No further formulas (than the ones in the addressed rewriting rule) with other prefixes than before and after are defined for a rewriting rule.
- \( R \rightarrow R', N \rightarrow R' \) and \( P_1 \rightarrow R' \) are jointly surjective.
- The set of constraints on \( R' \) is not conflicting.

Rule 1

This deduction rule expresses that a graph pattern, which was not present in the input model, also does not exist in the output model. Though, this is only the case if the following application is true: if a graph pattern \( N \) exists in the output model then it exists also in the input model. Formally, this is described as follows:

- Given: a rewriting rule \( s \) with the assertions attached to it:
  - \( a_1 : \text{before}(s) \) None \( N \)
  - \( a_2 : \text{at}_{\text{succ}}(s) \) ForOne \( L \rightarrow R \)

- sufficient condition: there exists a morphism \( N \rightarrow L' \) for every possible \( L' \) according to the DPO diagram:

  \[
  \begin{array}{c}
  L \xleftarrow{(1)} K \xrightarrow{(2)} R \\
  L' \xleftarrow{s} K' \xrightarrow{r'} R' \\
  N
  \end{array}
  \]

- conclusion: \( \text{after}(s) \) None \( N \) is true

Rule 2

The second rule is about the presence of morphisms. A morphism that is present in the input model between two graph patterns is also present in the output model if the two patterns themselves are still existent in the output model.

- Given: a rewriting rule \( s \) with the assertions attached to it:
  - \( a_1 : \text{before}(s) \) Any \( P_1 \xrightarrow{m} P_2 \)
  - \( a_2 : \text{at}_{\text{succ}}(s) \) ForOne \( L \rightarrow R \)

- sufficient condition: according to the DPO diagram

  \[
  \begin{array}{c}
  L \xleftarrow{m} K \xrightarrow{r} R \\
  R' \xrightarrow{p} P_1 \xrightarrow{m} P_2
  \end{array}
  \]

there exists a morphism \( P_1 \rightarrow R' \) such that:
\[ P_1 \rightarrow P_2 = (P_2 \rightarrow R') \circ m \]
\[ \forall c \in AC_{P_2} \Rightarrow AC_{R'} \vdash (P_2 \rightarrow R')(c) , \]
where \( AC_{P_2} \) is the set of constraints defined on pattern \( P_2 \)

- conclusion: after(s) Any \( P_1 \rightarrow P_2 \) is true

**Rule 3**

This rule presents a graph pattern \( P_1 \) contained in input model, rewriting rule and output model and an implication \( P_1 \xrightarrow{m} P_2 \) being true for the input model. After the rewriting rule the implication is still true if \( P_2 \) has a morphism to an instance of the output model. A second DPO is made.

- Given: a rewriting rule \( s \) with the assertions attached to it:
  - \( a_1 : \text{before}(s) \) Any \( P_1 \xrightarrow{m} P_2 \)
  - \( a_2 : \text{at\_succ}(s) \) ForOne \( L \rightarrow R \)

- condition 1: according to the following diagrams, which are composed for every possible \( R' \) object (1) and every \( L' \) object (2),

\[
\begin{align*}
L & \leftarrow K \rightarrow R \quad P_1 \\
L' & \leftarrow K' \rightarrow R' = P_2
\end{align*}
\]

- \( P_1 \rightarrow K' \) exists, such that \( (P_1 \rightarrow R') = (K' \rightarrow R') \circ (P_1 \rightarrow K') \)
- \( P_2 \rightarrow L'' \) and \( L' \rightarrow L'' \) are jointly surjective
- \( e_4 \circ p = e_3 \circ m \)

- condition 2: according to the DPO diagram

\[
\begin{align*}
L' & \leftarrow K' \rightarrow R' \leftarrow e_2 \quad P_1 \\
L'' & \leftarrow K'' \rightarrow R'' \leftarrow e_6 \quad P_2
\end{align*}
\]

there exists a morphism \( P_2 \rightarrow R'' \) for every possible \( R'' \) object such that:
- \( e_5 \circ e_2 = e_6 \circ m \)
- \( \forall c \in AC_{P_2} \Rightarrow AC_{L''} \vdash (P_2 \rightarrow L'')(c) , \)
  where \( AC_{P_2} \) is the set of constraints defined on pattern \( P_2 \)

- sufficient condition: condition 1 and 2 are true
- conclusion: after(s) Any \( P_1 \rightarrow P_2 \) is true

**Rule 4**

Rule 4 is related to rule 1. If the application of a rewriting rule does not modify a pattern, then also the repeated application does not change it.

- Given: a rewriting rule \( s \) with the assertions attached to it:
  - \( a_1 : \text{before}(s) \) None \( N \)
  - \( a_2 : \text{at\_succ}(s) \) ForEach \( L \rightarrow R \)
• sufficient conditions of rule 1
• conclusion: after(s) None N is true

Rule 5
Rule 5 generalizes rules 2 and 3 for the exhaustive application.

- Given: a rewriting rule s with the assertions attached to it:
  - $a_1 : \text{before}(s) \text{ Any } P_1 \rightarrow P_2$
  - $a_2 : \text{at}_{\text{succ}}(s) \text{ ForEach } L \rightarrow R$

- sufficient conditions of rule 2 or rule 3
- conclusion: after(s) Any $P_1 \rightarrow P_2$ is true

This section is based on [ALL10].

4.2 Assertion Sets

This section presents the assertion sets of the rewriting rules of the transformation patterns. For each transformation there are requirements on the input model and the assertions expressing the rewriting rules.

The assertions contain several graph pattern names. The corresponding diagram can be found in figure 4.1.

This and the following section should have been supported by the verification plug-in of VMTS. (See also chapter 6)

Table 4.1 presents the initial assertion sets of all the introduced patterns. For each rewriting rule there is one dynamic assertion, as described in the beginning of this chapter. A graph pattern of the source model is transformed to a graph pattern of the target model. All rewriting rules are executed exhaustively which means all LHS matches that can be found in the input model are replaced by the RHS.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping Pattern</td>
<td></td>
</tr>
<tr>
<td>at\text{succ}(\text{StateItemMapping Rule})</td>
<td>ForEach $P_3 \rightarrow P_4$</td>
</tr>
<tr>
<td>at\text{succ}(\text{StateElementMapping Rule})</td>
<td>ForEach $P_5 \rightarrow P_6$</td>
</tr>
<tr>
<td>at\text{succ}(\text{HistoryMapping Rule})</td>
<td>ForEach $P_7 \rightarrow P_8$</td>
</tr>
<tr>
<td>at\text{succ}(\text{sStateMapping Rule})</td>
<td>ForEach $P_9 \rightarrow P_{10}$</td>
</tr>
<tr>
<td>at\text{succ}(\text{TransitionMapping Rule})</td>
<td>ForEach $P_{11} \rightarrow P_{12}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refinement Pattern</td>
<td></td>
</tr>
<tr>
<td>at\text{succ}(\text{Refinement Rule})</td>
<td>ForEach $P_1 \rightarrow P_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Abstraction Pattern</td>
<td></td>
</tr>
<tr>
<td>at\text{succ}(\text{Abstraction Rule})</td>
<td>ForEach $P_2 \rightarrow P_{13}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flattening Pattern</td>
<td></td>
</tr>
<tr>
<td>at\text{succ}(\text{TransitionRedirecting Rule})</td>
<td>ForEach $P_{14} \rightarrow P_{15}$</td>
</tr>
<tr>
<td>at\text{succ}(\text{CompositeFlattening Rule})</td>
<td>ForEach $P_{16} \rightarrow P_{17}$</td>
</tr>
<tr>
<td>at\text{succ}(\text{SuperstateRemoving Rule})</td>
<td>ForEach $P_{18} \rightarrow P_{19}$</td>
</tr>
</tbody>
</table>

Table 4.1: Initial Assertion Sets for the Transformation Patterns

Additionally, an extended assertion set is created which contains the constraints for the
input model. In practice, the input model has to be checked. If it holds all the constraints and the offline verified transformation is executed, then all proved properties hold in the output model. Another possibility is to realize the constraints in the metamodel of the input model. In that case, no incorrect instances can be created in a transformation tool like VMTS. The assertions in table 4.2 hold before every first rule of the CFs of all the transformation patterns. Besides, an explanation is given that exemplifies the constraint.

### 4.3 Verified Properties

The properties to be verified are also expressed as assertions. Their prefix is \textit{after(lastRule)} because they shall hold in the model after the last rewriting rule of the CF. It is the output model.

With the deduction rules the basic assertions are propagated through the model and the
Input-Model Constraints

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>before(firstRule) : None P6</td>
<td>No UML elements exist in the input model</td>
</tr>
<tr>
<td>before(firstRule) : Exist P19</td>
<td>The most outer frame, the state machine, exists. It is a StateItem.</td>
</tr>
<tr>
<td>before(firstRule) : Any P3 → P20</td>
<td>Every super-state has at least one sub-component.</td>
</tr>
<tr>
<td>before(firstRule) : Any P3 → P23</td>
<td>Every sub-state machine has a first state.</td>
</tr>
<tr>
<td>before(firstRule) : Any P5 → P21</td>
<td>Each stateElement has an outgoing transition, otherwise it would be a final state.</td>
</tr>
<tr>
<td>before(firstRule) : Any P5 → P22</td>
<td>Each stateElement has an incoming transition, otherwise it would be a start state.</td>
</tr>
<tr>
<td>before(firstRule) : Any P21 → P2</td>
<td>If a stateElement has an outgoing transition, it also has an incoming transition.</td>
</tr>
<tr>
<td>before(firstRule) : Any P22 → P2</td>
<td>If a stateElement has an incoming transition, it also has an outgoing transition.</td>
</tr>
</tbody>
</table>

Table 4.2: Extended Assertion Set for the Transformation Patterns

Properties to be verified can be derived from them. Table 4.3 presents the properties which are verified for the transformation patterns.

**Mapping** The mapping transformation is actually not a typical VMTS transformation. The input model is passed through the CF and only the attributes (names) are changed. The structure of the model does not change by any rewriting rule. By this preserved structure is proved. In table 4.3 the Terminates formula is referenced with the mapping pattern. Generally, graph transformation termination is undecidable [PB10]. However, neither the rewriting rules nor the CF contain loops and therefore termination can be assumed.

**Node Abstraction and Refinement** After changing the structure through transformation it should be assured that no states are created that cannot be left or reached. Therefore the formulas given in table 4.3 have to be proved. Every stateElement shall have at least one incoming and one outgoing transition. These properties are given in the beginning of the execution. Deduction rule 5 transfers them to the end of the CF which consists of one rewriting rule. In the following is shown that the deduction rule can be applied.

With \( L = P1, R = P2, P_1 = P5 \) and \( P_2 = P22 \) the condition of rule 2 is true, because every possible \( R' \) contains \( P2 \). Since \( P5 \) and \( P22 \) are subgraphs of \( P2 \), they are also both part of every possible \( R' \). No additional constraints are defined and therefore rule 5, which uses the condition of rule 2 (or rule 3), can be applied. In the same manner, this can be shown for the outgoing transition \( (P21) \). The transition refinement rule is thereby proved for the two given properties.

For the node abstraction rule the condition of deduction rule 2 cannot be fulfilled. Instead,
<table>
<thead>
<tr>
<th>Mapping Pattern</th>
<th>Refinement Pattern</th>
<th>Node Abstraction Pattern</th>
<th>Flattening Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>at_{succ}(Mapping Rule) : Terminates</td>
<td>after(Refinement Rule) : Any P5 → P21</td>
<td>after(Refinement Rule) : Any P5 → P22</td>
<td>after(SuperstateRemoving Rule) : None P28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>before(SuperstateRemoving Rule) : None P26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>before(SuperstateRemoving Rule) : None P27</td>
</tr>
</tbody>
</table>

| | | | |
| | | | No hierarchy exists anymore. |
| | | | Before removing a super-state it should not have outgoing transitions anymore. |
| | | | Before removing a super-state it should not have incoming transitions anymore. |

Table 4.3: Assertions of Verified Properties of the Transformation Patterns

rule 3 can be applied and the transformation proved. Generally it can be said, that more information about the input model, which means having a bigger initial assertion set, make it easier to derive truth of properties of the output model.

**Flattening** For the flattening transformation three properties are aimed to be verified. The first one states that after the transformation no super-states, except from the state machine, shall exist anymore. The other two properties assure that super-states are free of incoming and outgoing transitions before the last rule is executed and finally deletes the super-states. In [ALL10] this transformation in proved. The initial assertions can be propagated through the CF by the given deduction rules.

**Remark** The author of [Asz12] mentions that so called domain experts could extend the initial assertion set with their knowledge about the system by other true formulas. If the set is extended and more deduction rules are developed, far more properties can be verified formally. ADL is designed to be expendable and it is possible to add such properties from experts. [ALL10] [Asz12]

This offline verification does not have to be repeated for other input and output models. All properties that hold for a transformation will also hold for any of its input-output model pairs.

The verification depends on the metamodels and would have to be repeated for other ones.
5 Case Study – Use of Transformation Patterns

This case study presents a transformation of a sample state machine of a real Airbus system. Specifications are written in English language, e.g., in plain text, but in few documents a model is attached. This example handles the classic mode of the Light Control in an aircraft cabin.

In classic mode, the flight attendant has the opportunity to select one of four lighting modes (NIGHT, DIM1, DIM2 or BRT) or unselect all of them (OFF). Additionally, if there is an emergency (smoke, low pressure or low airflow), the system changes, independently of the previous lighting, to the according emergency state. This behavior is described on system level, the corresponding state machine can be seen in figure 5.1. An initial state is not assigned because it can be defined for each instance variably.

This model describes approximately one quarter of the user interfaces related with the light control. Around one hundred instances of this model will be comprised in one airplane. Additionally to the light control, there are state machines in audio control, for startup behavior and software loading. 15 - 20% of the cabin management systems are state machines. That is why the specialization of this work to state machines has a reasonable background. [Kil13]

The project was initiated because software developers have to analyze the system specification in order to generate the software specification. If there is a specification given as model, the developer creates another state machine manually. It is describing the software and therefore it is different from the system model. Before, when working with the textual requirements, the developer had to reference the system requirements in his software requirements. Certainly, this also has to be done, somehow, with models. By manual creation of the software model it is not known if an illegal transformation took place and, e.g., a state is missing. By documenting the changes in a transformation script, like a QVT source code, the changes are traceable. The suggested, verified, patterns can be used. However, using a transformation script already helps to track system definitions. For example, when using a script, requirements cannot be deleted accidentally like in a manual transformation.

5.1 Transformation

The software model of the above mentioned system is given in figure 5.2.

All five (four + OFF) lighting modes are divided into two states. An intermediate state is added which indicates the fading of the light. Furthermore, the emergency states are combined to one state because an independent software module shall be developed. The specific behavior shall not be not shown in the normal operation mode model. Additionally, a new state, SHUT_DOWN, is found in the software state machine.
Node Refinement

The states in the Normal Illumination super-state are transformed with the Refinement Pattern. The upper right rule of figure 3.6 visualizes the transformation. The author of [Nov] calls it Node Refinement, while the example (chapter 3.3) of this work presents the Transition Refinement.

In order to only transform the states inside the super-state, the corresponding mapping of their context is comprised in the \texttt{when} clause.

The new element will have the extended name with \texttt{_IM} for “intermediate”.

The incoming transitions of a state in the super-state are mapped to incoming transitions of the corresponding intermediate state. Likewise, the outgoing transitions of this state are mapped to the outgoing transitions of the second state. In between there is one transition from the intermediate state to the other one. The whole state machine is surrounded by a \texttt{stateItem} with the aim to name the most general instance.

The following code is the beginning of the transformation script.

\begin{verbatim}
transformation LightControlSystem (input: sysml, output: uml) {

  top relation IntermediateState {
    n : String;
    checkonly domain input node : sStateElement {
      name = n,
      context = c1 : sStateItem {} }
    };
    enforce domain output src_node : uStateElement {
      name = n + '_IM',
      context = c2 : uStateItem {} }
    };
    when {
      StateItemMapping(c1,c2);
    }
  }

  top relation finalState {
    n : String;
    checkonly domain input node : sStateElement {
      name = n,
      context = c1 : sStateItem {} }
    };
    enforce domain output snk_node : uStateElement {
      name = n,
      context = c2 : uStateItem {} }
    };
    when {
      StateItemMapping(c1,c2);
    }
  }

  top relation NodeRefinement {
    checkonly domain input node : sStateElement {};
    enforce domain output src_node : uStateElement {};
    enforce domain output snk_node : uStateElement {};
    enforce domain output e : sTransition {
      left = src_node,
      right = snk_node,
      name = src_node.name + '_to_ ' + snk_node.name,
      context = src_node.context }
    };
    when {

32
\end{verbatim}
IntermediateState(node, src_node);
finalState(node, snk_node);
}
}

top relation incomingTransition {
  n : String;
  checkonly domain input e1 : sTransition {
    left = leftI : sStateElement {},
    right = rightI : sStateElement {},
    name = n,
    context = c1 : uStateItem {};
  };
  enforce domain output e2 : uTransition {
    left = leftO : uStateElement {},
    right = rightO : uStateElement {},
    name = n,
    context = c2 : uStateItem {};
  };
  when {
    finalState(leftI, leftO);
    IntermediateState(rightI, rightO);
    StateItemMapping(c1, c2);
  }
}

top relation outgoingTransition {
  n : String;
  checkonly domain input e1 : sTransition {
    left = leftI : sStateElement {},
    right = rightI : sStateElement {},
    name = n,
    context = c1 : uStateItem {};
  };
  enforce domain output e2 : uTransition {
    left = leftO : uStateElement {},
    right = rightO : uStateElement {},
    name = n,
    context = c2 : uStateItem {};
  };
  when {
    finalState(leftI, leftO);
    IntermediateState(rightI, rightO);
    StateItemMapping(c1, c2);
  }
}

top relation StateItemMapping {
  mm : String;
  checkonly domain source se1 : sStateItem {
    name = mm
    context = c1 : sStateItem {};
  };
  enforce domain target se2 : uStateItem {
    name = mm
    context = c2 : uStateItem {};
  };
  StatemachineMapping(c1, c2);
}

top relation StatemachineMapping {
Node Abstraction

The emergency illumination is represented, in the system model, by three states and written requirements attached to the model. It was decided to implement an own module and replace it in the complex state machine by a single state. Like this, the state machine stays human readable. If one wanted to model also the textual requirements, the state machine would have got unclear.

This transformation seems to be a node abstraction but in fact the pattern cannot be applied since it is optimized for sequential states. Still, the basic idea for the transformation code is taken from that pattern. The three states have to be matched in the input model and mapped to only one state in the output model:

```java

mn: String;

checkonly domain source se1: sStateItem {
    name = mn
}

enforce domain target se2: uStateItem {
    name = mn
}

--no context and when statement because it is the most general concept
}

...

Transition Mapping

After all, the transitions from the normal mode to the emergency state are not mapped, yet. This is done by two instances of the Transition Mapping Pattern, adapted to this case. Each instance is responsible for one direction.

```
New State

The SHUT_DOWN state is a state that is completely new in the software model. In the system model it was not desired to entirely turn off the system. Regarding the question if this is an allowed transformation, the answer is "no". Introducing a new behavior might have good reasons, but is the task of the system designer. Besides, there is no pattern for it. Trying to use the Refinement Pattern is first of all against the intention of the pattern, which is introducing a subsystem in a part of the model. Secondly, it always has to flow to the same unchanged state. In contrast, the SHUT_DOWN state finishes the system, meaning it is a final state. In this case, the software designer made an illegal transformation.
5.2 Discussion

The case study demonstrates how practicable the introduced transformation patterns are and which additional problems appear. The main part of the model can be transformed with the patterns. It cannot be estimated, if this is also the case for all other Airbus models. In order to have a realistic result, it cannot be worked on sample models, only on the originals. Unfortunately, MDA is not yet sufficiently developed in this Airbus department and no more models are available for the system.

Still, there will always be parts which cannot be transformed by a pattern. In this case study, the reason is readability. Models are still read by humans and a transformation can intent clarity, additionally to platform specifications. Although model-to-code transformations are in a good state of development and MDA might change software development with the time, legibility of models is an important factor for a model’s success. For instance, the validation of a model still has to be done by the developer, a human that interprets the content by watching the model. The changes to be made in this case can be still expressed in QVT. A model can also be transformed if there is no pattern for it. Even if there is no verification of this part, it can be still a legal transformation.

The used system and software models are realistic models. They are taken from the corresponding specification documents. As mentioned in the introduction of this chapter, state machines describe a significant part of the system and the patterns are specialized to them. However, the patterns can also be used for other kinds of models. Surely, the element types have to be renamed, but especially if a modeling language has a similar structure, the pattern can be taken as it is. Nevertheless, further verified patterns for other model types would help to make a whole system transformation practicable.
Figure 5.1: System State Machine in Classic Lighting Mode

Figure 5.2: Software State Machine in Classic Lighting Mode


6 Conclusion

6.1 Results

In this project work four model transformation patterns are introduced. They were chosen because they deal with common tasks in state machine transformations. The aim was to find means that help the Airbus software developers to convert system models from PIM level to software models in PSM level. Until now, these transformations are made manually. It is suggested to use QVT for specifying the relations between two models. QVT scripts can be processed by transformation tools but also without using a tool they are useful to document the changes made from system to software level.

The introduced transformation patterns are design patterns that help to generate typical PIM to PSM transformation scripts. They can be used as they are but in most cases they will be combined with additional code. This is usual for design patterns like in other languages such as Java.

Here, the main focus is on state machines. They cover a big part of the Airbus system. Also other model types can be treated with the same patterns. They would have to be slightly adapted but the basic idea stays the same. This could be done in a future work. In the case study 65% of the model could be transformed with only two patterns. Even if it is not a representative field study, it can be seen that these few patterns can influence a big part of the system.

This work covers offline verification. This means that a defined transformation is verified and any valid input produces an output with assured functional properties. Practically this means that if a system model has an approved transformation to a software model, the designer can make changes on the system model. For the software model no further investigation has to be done. Only the transformation script has to be executed and the updated software model is created. This helps to save development costs since an expensive re-analysis is avoided.

Graph-rewriting is used to formally describe and verify model transformations. Furthermore, a system exists that realizes this approach in software. It is developed by Hungarian scientists and is called VMTS. The system has many helpful functions linked in the user interface or mentioned in papers. Unfortunately the documentation and support is not as developed as the system and most features cannot be used instinctively. The results of this work suffer from this fact. For example, the conversion from QVT to the VMTS modeling and transformation approach could not be validated by the software as it is described in [LLVC06]. However, in the beginning of the project VMTS was very promising and therefore it was chosen to work with. The research on formal verification of model transformations is recent. For example, a dissertation on it and the verification plug-in was published less than one year ago [Asz12]. Unfortunately, this plug-in is another example for a promising feature that could not be used because of the reasons mentioned above.

The verification of QVT patterns can be evaluated as complex and inefficient. The patterns in this work are verified but since they might be adapted to special systems or extended by other code the verification does not cover the whole transformation. Additionally, it has
to be mentioned that QVT and VMTS base on two diverse logics. The VMTS software has a plug-in for the transformation of these logics, based on the VMTS transformation logic. However, as mentioned before, it was not usable in this work. Still, the procedure of verification is shown here and can be repeated for any transformation. Furthermore, the process was automated recently by the author of [Asz12].

6.2 Future Work

As already mentioned above, system specifications at Airbus are not yet completely model based. Furthermore, there will not be a clear restructuring. Hybrid documentation will be used instead. This means that in addition to the models there will be textual requirements, also. The SHUT_DOWN state in the case study was such a textual requirement which the software developer realized in the software model. In terms of model transformation this was illegal but in terms of specification it was valid. In a future work this problem has to be analyzed.

In order to apply the results in Airbus environment, a tool chain has to be found. Most models are created in Artisan Studio. Exporting them from there and importing them to a suitable QVT processor is a suggested future task. The basic idea of VMTS is good for the reasons explained before and making the system suitable for productive work can be a hard but worth future work.

Concerning the transformation patterns, future work can be, introducing more of them. Also patterns for further model types might be useful as well as the adaption of the current ones. The transformation between more diverse model types can be a task, as well as more complex metamodel types which have to be adapted. The combination of several input models and splitting to several output models is a feature of QVT that was not handled in this work. Nevertheless it is an interesting topic in this field.
Bibliography


Declaration

I, Anna-Maria Liebhoff, solemnly declare that I have written this project work independently, and that I have not made use of any aid other than those acknowledged in this project work. Neither this project work, nor any other similar work, has been previously submitted to any examination board.

Buxtehude, September 4, 2013