JOHANNES KEPLER
UNIVERSITY LINZ
JKU

TNF
Faculty of Engineering
and Natural Sciences

Constraint Generation and Partial Fixing for UML Models through Transformation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements
for the academic degree

Bachelor of Science

in

INFORMATIK

Submitted by
Stefan Luger

At the
Institute for Systems Engineering and Automation

Advisor
Univ.-Prof. Dr. Alexander Egyed M. Sc.

Co-advisor
Dipl.-Ing. Andreas Demuth

Linz, August 2013

Sworn Declaration

“I hereby declare under oath that the submitted Bachelor’s thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.”

Linz, January 9, 2014

"It has certainly been true in the past that what we call intelligence and scientific discovery have conveyed a survival advantage. It is not so clear that this is still the case: our scientific discoveries may well destroy us all, and even if they don't, a complete unified theory may not make much difference to our chances of survival."

Stephen Hawking

Abstract

Working with complex models not only requires knowledge and skills to design them, but more importantly, change triggered to model elements by humans may violate the well-formedness or semantic design rules. Especially, semantic relationships within Unified Modeling Language (UML) models are complicated to preserve. Model-driven engineering standards such as the Object Constraint Language (OCL) support the necessary consistency checking. However, the manual generation of such constraint expressions and in advance the elimination of inconsistencies does not suit our needs. In this thesis, an incremental model manipulation approach (achieved through the ATLAS Transformation Language) is used to present an application for both traditional model transformation and constraint-driven modeling with the goal to eliminate existing inconsistencies (as far as possible). During the process of the transformation, constraints are generated to validate the model with - all in the context of the UML. In particular, 9 different constraint-driven scenarios, supporting class, sequence and statemachine diagrams/models, were developed. To illustrate a successful transformation process, the updated model is validated via the OCL subsequently. Model designers in the needs of consistency checking and constraint generation benefit from the automatic and incremental execution provided by the implementation.

Contents

List of Figures

Listings

Chapter 1

Introduction

During the last decade, the wide variety or even the absence of standards for legacy and newly built software made it more difficult when interconnecting those systems. Furthermore, distributed and embedded software-intensive systems are in the need of *Platform-independent Models (PIM)* and Platform-specific Models (PSM) to standardize component interfaces. *Model-driven Engineering (MDE)* [4–6] best practices overcome this problems via the introduction of modeling standards, speaking of *Uniform Modeling Language (UML)*, *Model Object Facility (MOF)* and many more. Since the early 2000's, the *Object Management Group (OMG)* plays a key role in terms of *Model-driven Software Development (MDSD)* including required standards.

Whereas the PIMs, e.g. UML, provide the necessary tools to model structure and behavior, domain specific challenges have to be solved. *Domain Specific Languages (DSL)*, e.g. *Object Constraint Language (OCL)* [7], rely on their metamodel to describe declarative semantics and constraints upon the model elements in context.

Besides DSLs, common transformation languages, such as *Atlas Transformation Language (ATL)* [8, 9] or *Query/View/Transformation (QVT)* [10], generate target models from source models through transformation. Transforming models focuses not only on generation and refactoring in general, but more importantly, considers preserving consistency, bidirectional synchronization as well as incremental execution while improving performance.

In this thesis, we propose a framework partially fixing UML models by taking advantage of the ATL. As presented in [11], Jouault et al. shows the clear advantages of ATL, which supports refining mode in-place transformations. In addition to UML model transformations, one would want to evaluate its correctness. For this purpose, given a set of consistency checking scenarios for the UML model, the OCL is used.

The thesis is organized as follows. Chapter 2 introduces the UML specification as well as the UML models under test, the ATL transformation possibilities and at last the OCL. A running example illustrates a model-to-model transformation in Chapter 3. Chapter 4 is divided into two sections. First, in Section 4.1, 9 constraint-driven scenario implementations are presented. For each of them as follows:

1. Initial design choices are discussed.
2. A detailed look into the ATL module implementation is shown.
3. The OCL expressions are validated to check whether the inconsistencies were fixed correctly.

Second, in Section 4.2, a comprehensive usage documentation lists the required steps in order to run the proposed framework. Chapter 5 discusses related work and puts the proposed framework in contrast. Chapter 6 sums up the thesis and discusses essential aspects for future work. In Appendix A, the complete source code is provided in pretty printed format.

Chapter 2

MDE Technologies

The subsequent main chapter of this thesis proposes a framework taking advantage of three different MDE languages. Thus for each language, a brief introduction/documentation is helpful. In detail, we will focus on purpose, architecture and API-related documentation about the ATL, OCL and UML. According to the high popularity and development activity within the rich Eclipse community and the OMG, those three languages are very well considered one of the state of the art technologies in current MDE. The UML serves as an object-oriented representation for software-intensive systems. Given an UML model of choice, the ATL covers the transformation behavior: generating a target model based on rules within the transformation module. As a side effect, constraints can be generated alongside the model-to-model transformation too. These constraints are formalized with the OCL and specify consistency checks related to their UML model element in context. Finally the generated OCL expressions can be used to validate the transformation's outcome, the target model.

2.1 UML

In [12], the OMG describes the UML in a short but detailed formalization:

> *"The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of softwarebased systems as well as for modeling business and similar processes."*

The UML Version 1.1 was adopted by OMG in 1997 [13], accepted with Version 1.4.2 as the ISO/IEC 19501 standard [14] with the current Version being UML 2.4.1 [13]. The UML provides a wide variety of diagrams driven by the field of application, grouping them into structural and behavioral diagrams. The implemented framework covers the most common class, sequence and statechart (mostly refered as statemachine in this thesis) diagrams. As the three of them contain overlapping characteristics (e.g. operations either correspond to messages or activities), they provide a rich set of inter-relationships. These shared characteristics are then refered as (in)consistencies, processed via the ATL transformation, and further are checked with the help of constraints through the OCL.

Having a standardized language for modeling software-intensive systems, the UML is human readable and thus based on the *Extensible Markup Language (XML)* format as *XML Metadata Interchange (XMI)* [12]. As a consequence of handling different metamodel architectures, the *Meta Object Facility (MOF)* [15] evolved to specify a metamodel standard architecture for the UML in the past. The UML2 metamodel used is built upon the *Eclipse Modeling Framework (EMF)* [1] to satisfy the needs when working with EMF-based plug-ins like *Papyrus* inside Eclipse framework. For more information please look up Subsection 4.2.1. Anyway, Papyrus provides a convenient looking graphical representation of UML models and further is used to build UML models conforming the EMF-based UML2 metamodel within Eclipse framework.

The following example UML models are derived from existing models in [16, 17] and were built at the *Institute for Systems Engineering and Automation* [2] at JKU Linz. They cover the needs for testing purposes of this work and will be referenced during implementation and discussion in Section 4.1 of this thesis. These models were edited to satisfy and/or violate specific consistency rules. Thus they will not necessary be complete or conform to their originally intended behavior.

- A simple (Class) Inheritance UML model is shown in Figure 2.1. It consists of both a class and sequence diagram. Class `A` is a subclass of `C` and only class `A` is represented as lifeline.
- The UML model in Figure 2.2 demonstrates a simple light switch. Obviously a light can be switched on and off. The model is used to violate mutual naming conventions

---

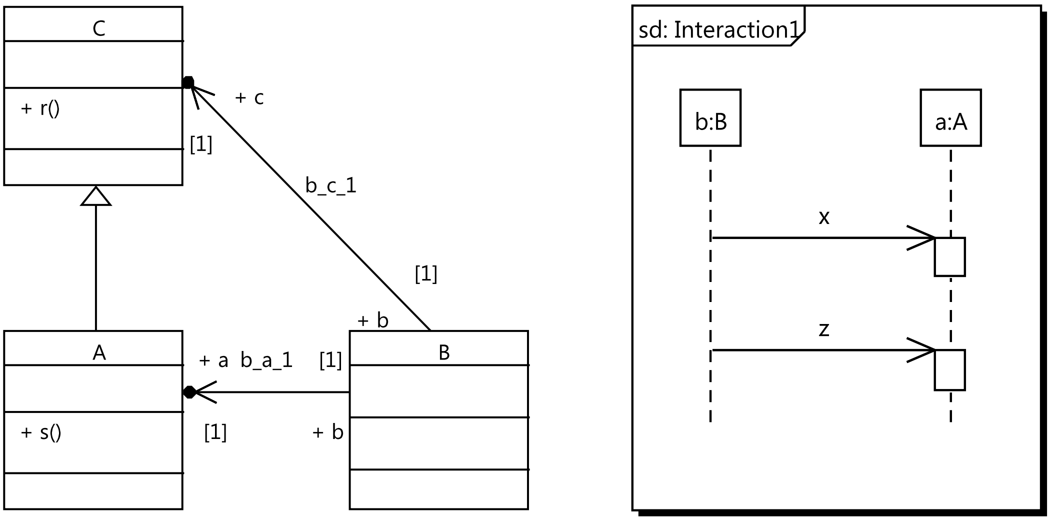
[1] Online at: `http://www.eclipse.org/modeling/emf/`.

[2] Online at: `http://www.jku.at/sea/content/e139529/e126342/e126449`.

between operation and message on purpose (e.g. message `deactivate` corresponds to an operation `deactivate()`).

- In order to focus on more complicated inter-relationships between diagrams, a video on demand system (shown in Figure 2.3) expands the set of models under test. An user is able to stream video content by selecting a movie of choice at any time.
- As shown in Figure 2.4 and Figure 2.5, the VOD UML model is extended by the class `Service` (which provides the pause feature) and an additional and more complex statemachine behavior for class `Streamer`.

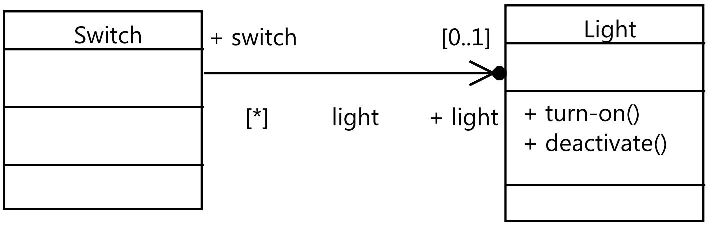
This models cover the needs for testing purposes of this work and will be referenced during implementation and discussion in Section 4.1 of this thesis. These models were edited to satisfy and/or violate specific consistency rules. Thus they will not necessary be complete or conform to their originally intended behavior.



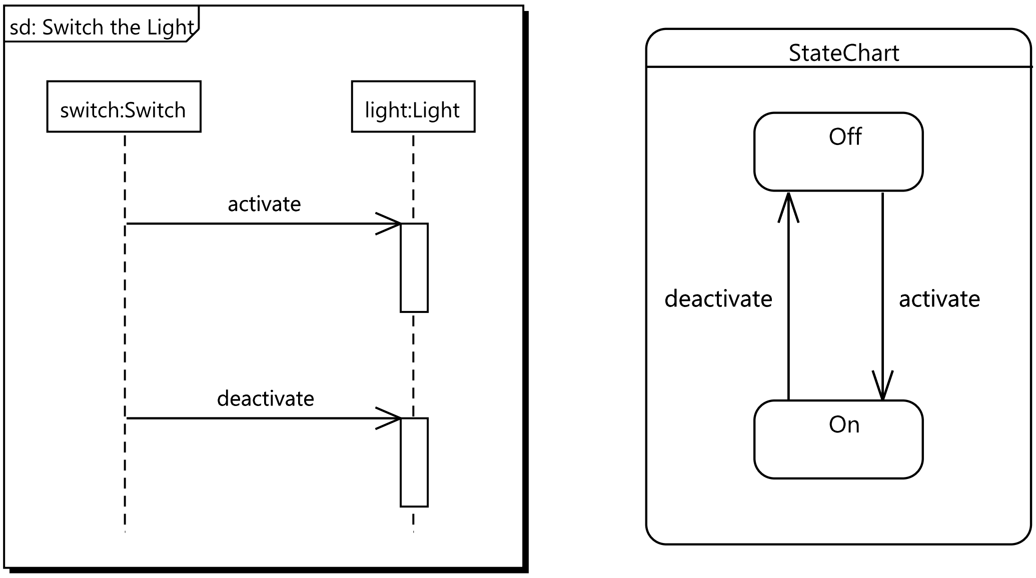
(a) Class Diagram.

(b) Sequence Diagram.

Figure 2.1: Inheritance UML Model Example.



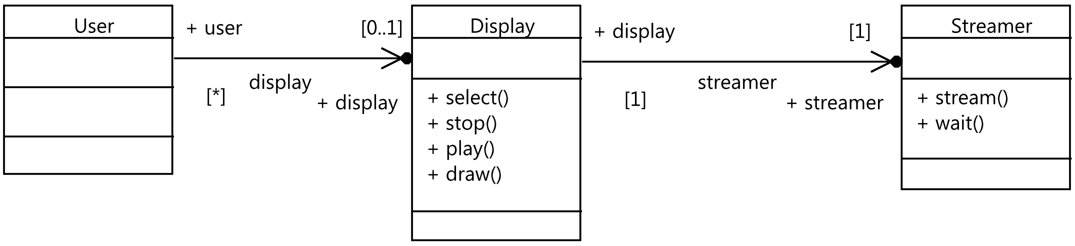
(a) Class Diagram.



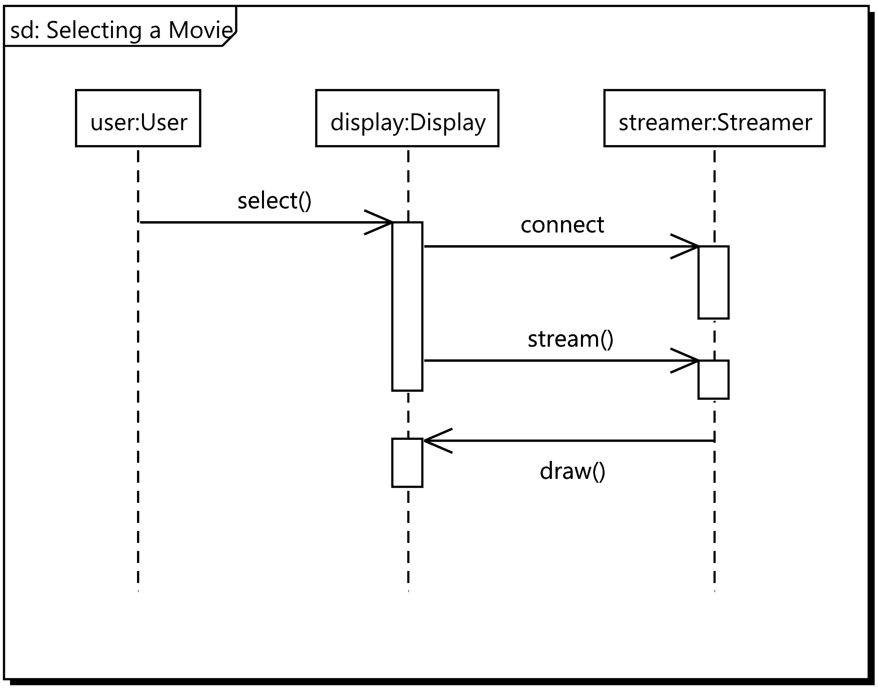
(b) Sequence Diagram.

(c) Light Statechart Diagram.

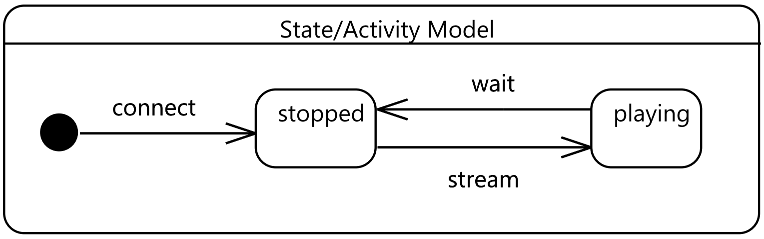
Figure 2.2: Light Switch UML Model.



(a) Class Diagram.

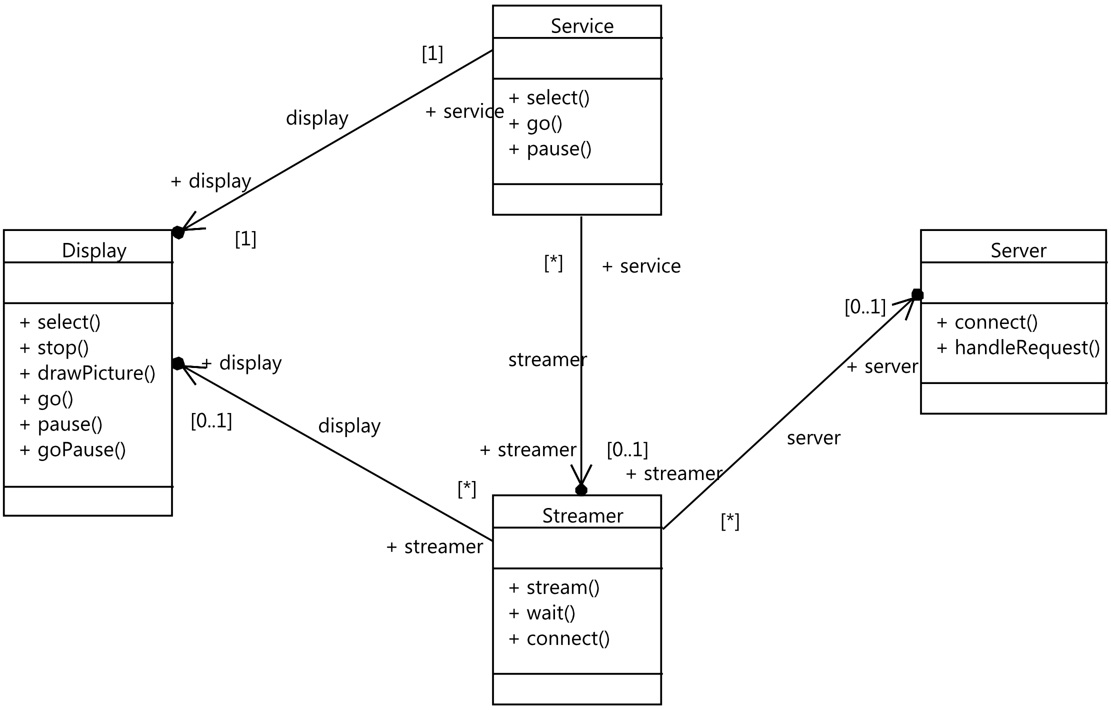


(b) Sequence Diagram.

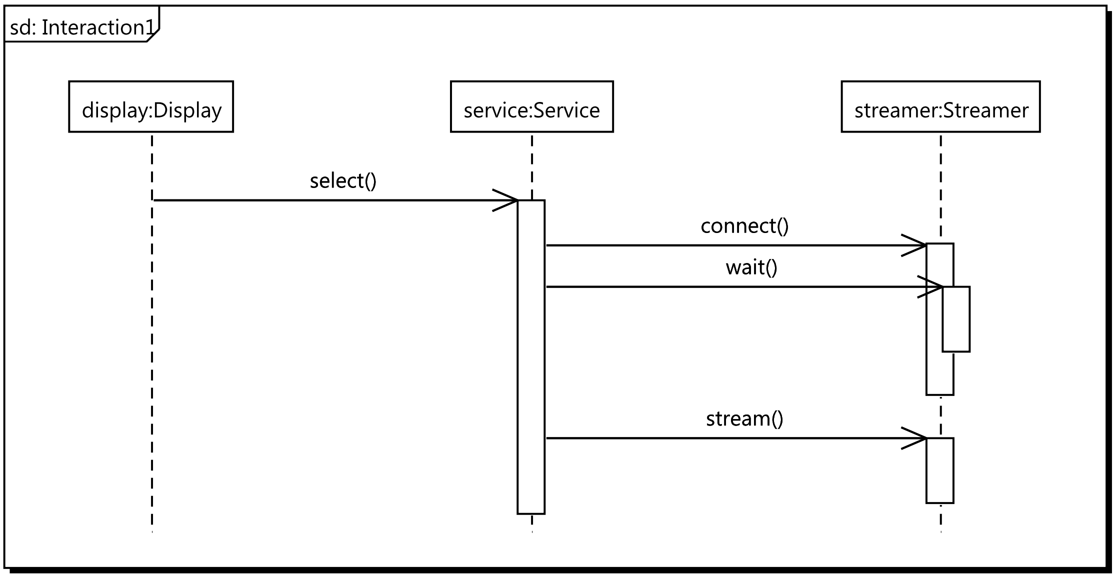


(c) Streamer Statechart Diagram.

Figure 2.3: VOD UML Model.

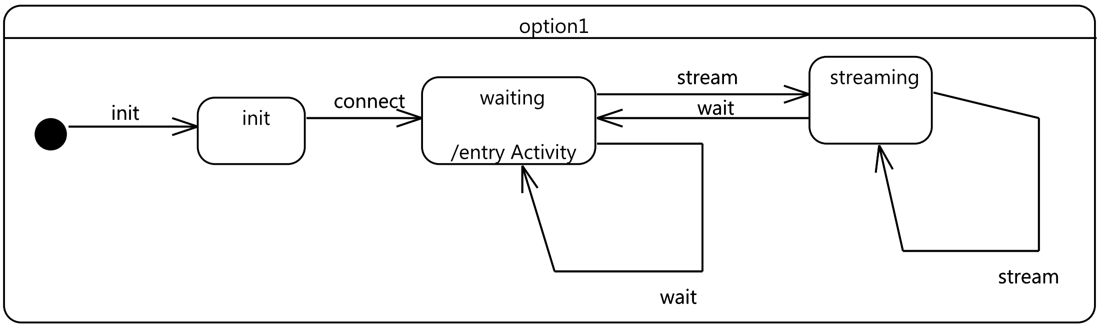


(a) Class Diagram.

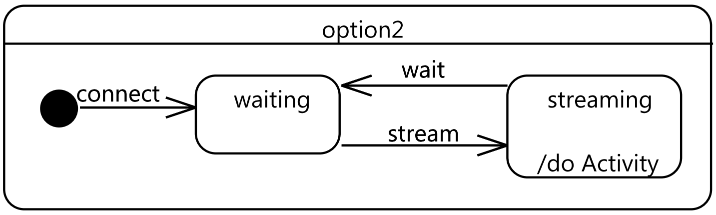


(b) Sequence Diagram.

Figure 2.4: VOD AR UML Model.



(a) Streamer Statechart Diagram Option 1.



(b) Streamer Statechart Diagram Option 2.

Figure 2.5: VOD AR UML Model continued.

2.2 ATL

The ATL, developed by Jouault et al. [8, 9], is a model transformation language built onto the Eclipse framework. The adaption of QVT aspects and continued support since 2006, when the ATL first was proposed, resulted in a highly anticipated language specification within the Eclipse *Model Development Tools (MDT)* [3] community.

ATL's concept is based on Figure 2.6, where a source model M_a is transformed into a target model M_b. Both models conform to their metamodels MM_a and MM_b, (e.g. UML). In order to perform the transformation, a transformation module M_t is required. The module conforms to its own metamodel again. According to the used MOF specification, all three metamodels conform to the same metametamodel MMM. Basically, the ATL provides the necessary language to describe such model transformations. The transformation process originating from one source model then results in one or more target models [1].

The architecture of the ATL contains an engine which compiles and executes the transformation module (a file with the ATL extension). The ATL2004, ATL2006 as well as ATL2010 compiler translates the source code to byte-code stored in the ASM file format (similar to an assembly language file). Finally the byte-code is executed by the ATL *Vir-*

---

[3] Online at: `http://www.eclipse.org/modeling/mdt/`.

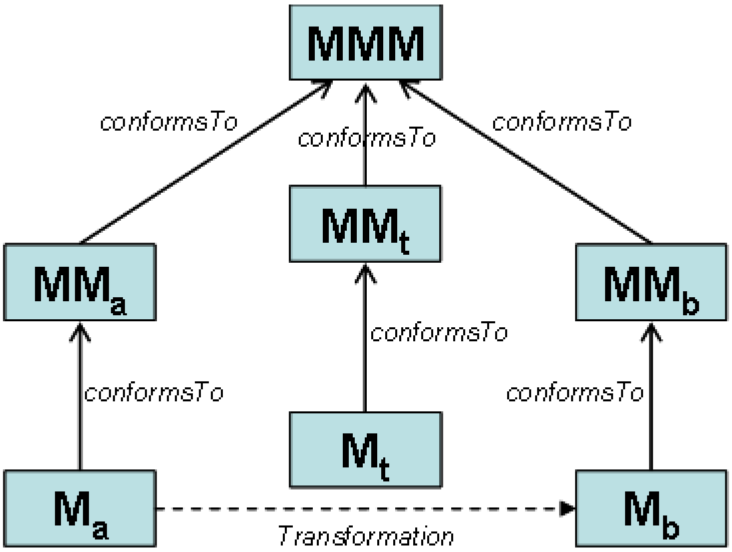


Figure 2.6: Model Transformation. [1]

tual Machine (VM) having its own instruction set. As the VM does not rely on the ATL itself, other VMs specializing on other languages can be developed to replace the regular VM [18].

For the following paragraphs we will give a brief introduction of the ATL specification. As an in-depth reference guide, the documentation at Eclipse wiki [2, 19] is suggested.

In general, the ATL is based on the OMG OCL. Introducing the data types, they are grouped into six different kinds with their root element being `OclAny`. `OclAny` behaves like the type `Object` in Java, where by default a base set of operations is provided. As the type names are relatively meaningful, we refer to Figure 2.7 and do not go further into detail.

Besides primitive type operations, collection type operations are worth mentioning. As one would expect, basic statements are supported: `iterate`, `collect`, `exists`, `select`, `includes`, `excludes`, `isEmpty`, `notEmpty`, etc.

Equally to the OCL, the ATL provides the keyword `self` which refers to the instance of such specific type. Just as a quick note on comments, they are expressed starting with two dashes followed by any characters: `-- comment text`.

Beginning at the very start of an ATL module, Listing 2.1 includes the most important statements required for an UML transformation. In the first line of code, an optional compiler is declared, which in this case happens to be the EMFTVM (EMF Transformation

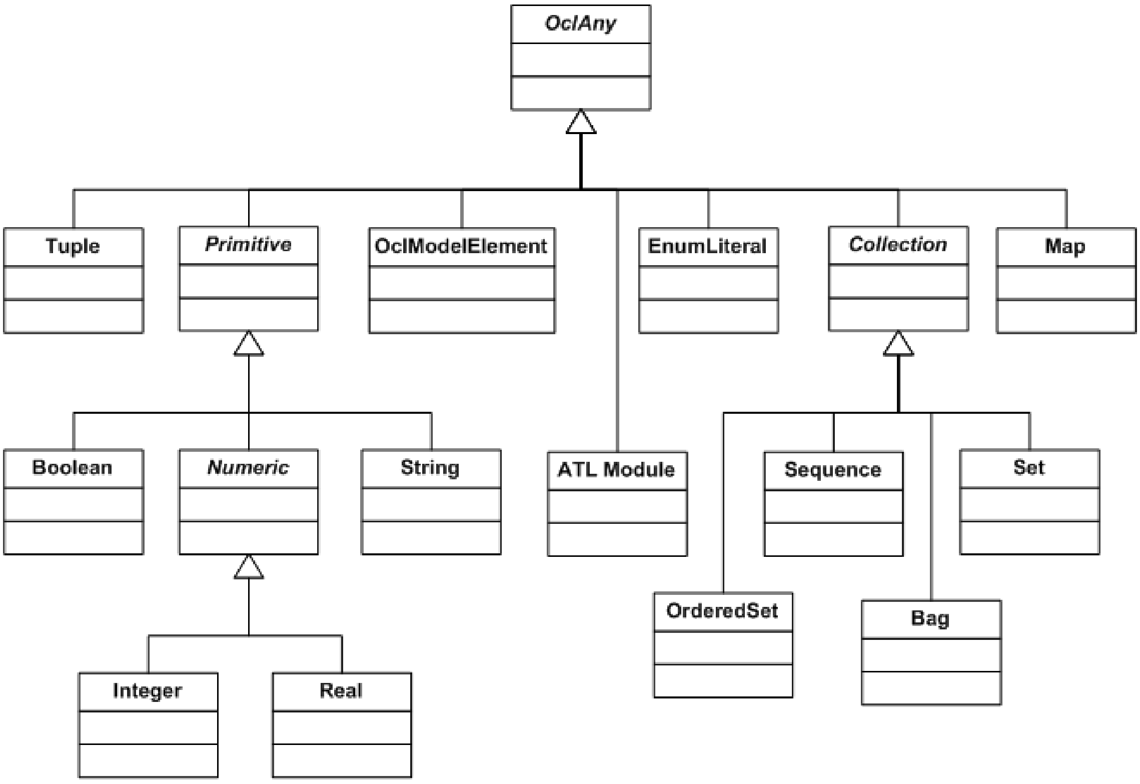


Figure 2.7: OCL Primitive Types. [2]

VM). We will introduce EMFTVM in the Section after the ATL documentation. In the second line, the Eclipse UML2 metamodel is declared. The actual transformation module starts in the third line with the keyword `module`, followed by the name which has to be equal to the file name (e.g. Seq2Class.atl). Concluding in the fourth line, three different characteristics are specified. After the keyword `create`, `OUT` represents the target model separated by its metamodel tag `UML2`. The keyword `from` signalizes the transformation to create a different target model. In contrast, one could replace it with *refining*, which would result in an in-place transformation. An execution through refining mode means that model elements which are not matched by any rule, stay unaffected. Whereas the normal execution mode produces a different target model, the refining mode just alters the source model. However, the source model `IN` and its metamodel `UML2` have to be declared anyway.

```
1  -- @atlcompiler emftvm
2  -- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML
3  module Seq2Class;
4  create OUT: UML2 from IN: UML2;
5  ...
```

Listing 2.1: ATL Module Header.

Subsequent ATL helpers are considered methods if compared to Java. They consist of optional parameters, a context and a return value. Helpers excel at navigational support through models and are programmed in declarative fashion only.

The most important components of ATL modules are rules. They are distinguished by the designator in front of keyword *rule*. A rule is divided into a maximum of four sections. The *from* section (source pattern) specifies the element type which has to be matched in order to trigger the rule. On the contrary, the *to* section (target pattern) encloses the actual transformation process. For each attribute of the target element, a new value based on the source model information can be assigned. Additionally, the optional *using* section allows the definition of variables in the scope of the rule, but only through declarative statements. The optional *do* section even allows imperative programming style, but one has to be aware that the *do* block is only executed when the *to* section has finished already.

Four different rules can be implemented:

- `rule`: A matched rule, in case no designator is specified, represents the most common rule used within transformations. Its name is derived due the fact the source pattern's type being matched onto an element with equal type. Matched rules may contain all of the previously explained sections.
- `lazy rule`: A lazy rule can be called from any statement within the module scope. However, the appropriate element type has to match the *from* section.
- `unique lazy rule`: This rule always returns exactly the same target element corresponding to a specific source element.
- `called rule`: Called rules allow optional parameters, but do not need a `from` section. Since they can be called from any statement within the module scope, and are often used for the generation of new element instances through the *to* and *do* sections.

Only within the *do* section, imperative programming mode is allowed. Usually, one would avoid the usage of imperative style because the complexity and readability might gets worse. Although, imperative style allows the usage of common programming statements such as `if` and `for` statements, declarative expressions are limited to nested `let` expressions. Nevertheless, primitive type operations, the assignment as well as collection statements are used in both programming styles.

To conclude with this section, let us quickly mention two of the key advantages the ATL has over QVT. Whereas ATL is considered hybrid and therefore allows a mixed (declarative and imperative) programming style, QVT separates both having to use either the one or other. Due to this, the ATL has the capability of designing rules more versatile [20, 21]. In [22], Amstel et al. measured the execution times of a simple transformation for ATL and QVT. Based on worst case scenario testing, the execution time for both languages differs significantly, resulting in ATL transformations having the better performance results. In conclusion, based on continued support, recently made improvements and the key features offered, the choice for ATL was rather obvious.

The base ATL Eclipse plug-in used for the practical work is listed in Subsection 4.2.1.

2.2.1 EMFTVM

As this work deals with EMF-based models and more complex transformation, and neither the ATL2004, ATL2006 or ATL2010 VM supports called rules with refining mode, the more specific EMFTVM [3] developed by Wagelaar et al. is introduced. The general ATL was presented in detail above, and due to the fact only the VM being replaced, only the changes and benefits of EMFTVM are presented. In contrast to the regular VM, where the byte-code is stored as XML format, EMFTVM uses EMF models for higher performance as shown in Figure 2.8. Further, helpers can make use of recursive calls, allowing for more complex implementations. Rules may associate with super rules or even are defined as abstract.

All in all, EMFTVM brings more functionality to model transformations and enhances performance dramatically when using in-place refinement mode.

EMFTVM is further discussed within Chapter 4.

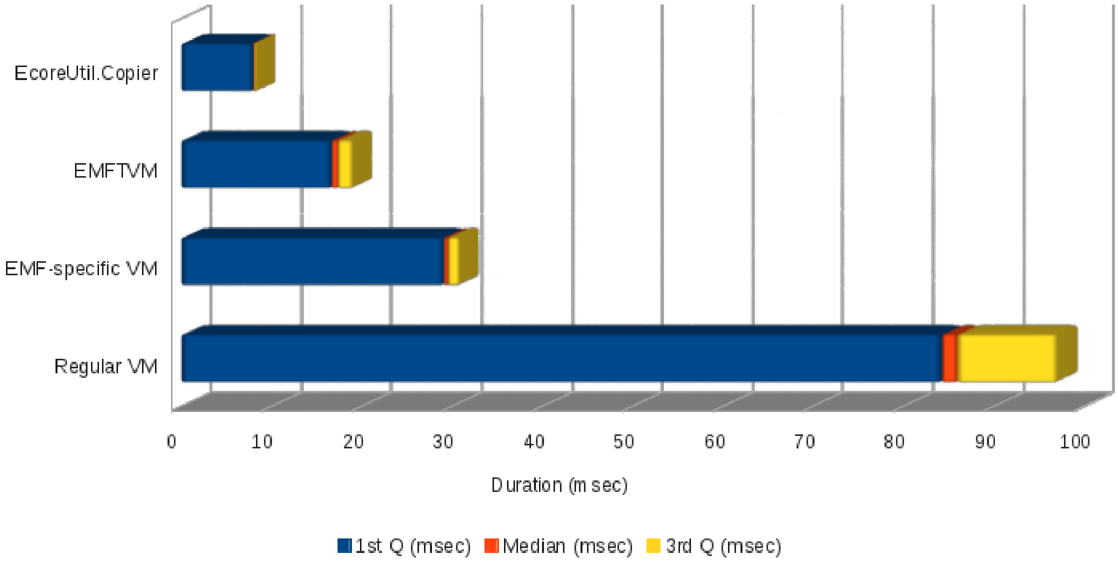


Figure 2.8: EcoreCopy Performance. [3]

2.3 OCL

The OCL, mostly used for checking consistency rules of models, excels for declarative and navigational purposes. It was developed at IBM and accepted as UML standard in the past. It conforms to any MOF metamodel. As the syntax for the OCL is more or less equal to the ATL, or because of the fact that the declarative ATL is derived from the OCL, the syntax is not introduced again. Nonetheless, a complete language documentation with examples can be found in [7].

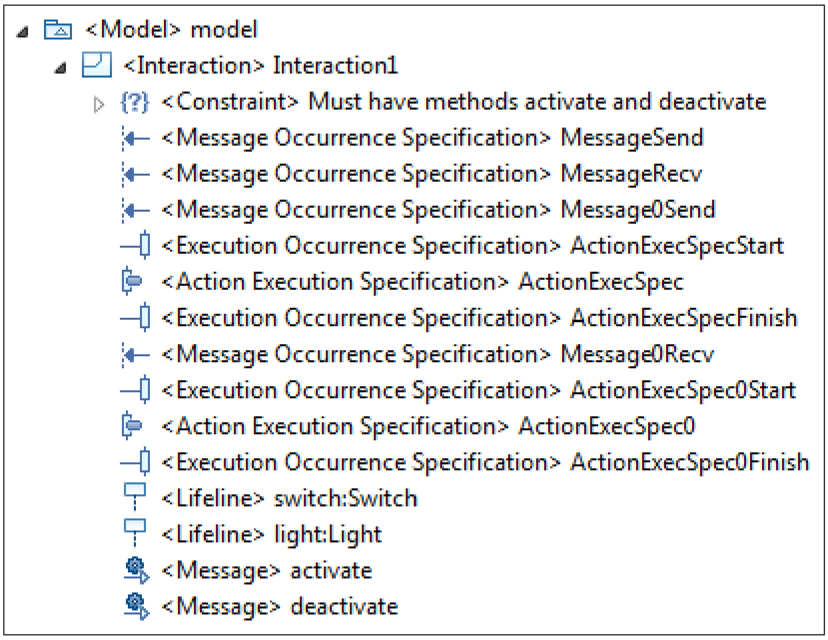
For the practical work, the OCL is used alongside the ATL transformation, where specific scenarios are partially fixed during transformation. For each scenario, constraints expressed in the OCL are then generated and validated with the help of Xtext OCL console (see Section 4.2).

Chapter 3

Motivating Example

To illustrate ATL transformations in form of a practical usage scenario, let us look at an example UML model transformation. For this purpose, a simple UML model shown in Figure 3.1 will serve as input model. The original model should be transformed to its corresponding class diagram. The direction, when transforming UML diagrams as a whole is important. One can notice that in the direction, with the sequence diagram being the input model, enough information is provided to create the class diagram respectively. In contrast, when trying to conduct a more complex transformation such as transforming a class to sequence diagram, it very well be nearly impossible. Indeed there is a reason why such diagrams are grouped as interaction or structural diagrams, since different diagrams provide a different focus on information presented. In fact, a class diagram does not contain any other information except messages, lifelines and the connectivity of lifelines through messages. In other words: messages represent operations, lifelines classes and associations give information about receiver and sender lifelines. But at the bottom line, no information on timely order is given. Thus multiple messages between lifelines would not make any sense at all, as we are not able to bring them in an order.

Having described the reason for the forthcoming transformation, we will directly step into code snippets, the first presented in Listing 3.1. As mentioned, we have to build associations in order to conform to the messages lifeline connections. With the help of additional helpers shown in Listing 3.2 and Listing 3.3, both the sender and receiver lifeline are provided. The last part of the helper `getAssociations` will then append all lifeline pairs for all messages within the input model.



- <Model> model
  - <Interaction> Interaction1
    - <Constraint> Must have methods activate and deactivate
    - <Message Occurrence Specification> MessageSend
    - <Message Occurrence Specification> MessageRecv
    - <Message Occurrence Specification> Message0Send
    - <Execution Occurrence Specification> ActionExecSpecStart
    - <Action Execution Specification> ActionExecSpec
    - <Execution Occurrence Specification> ActionExecSpecFinish
    - <Message Occurrence Specification> Message0Recv
    - <Execution Occurrence Specification> ActionExecSpec0Start
    - <Action Execution Specification> ActionExecSpec0
    - <Execution Occurrence Specification> ActionExecSpec0Finish
    - <Lifeline> switch:Switch
    - <Lifeline> light:Light
    - <Message> activate
    - <Message> deactivate

Figure 3.1: Light Switch UML Model.

```
helper def: getAssociations(): Sequence(OclAny) =
  let rcv: OclAny =
    thisModule.getReceiveLifelines() in
    let snd: OclAny =
      thisModule.getSendLifelines() in
      rcv -> iterate(i; assSeq: Sequence(UML2!"uml::Lifeline") = Sequence
          {} |
        assSeq.append(Sequence{i, snd -> at(assSeq.size() + 1)}));
```

Listing 3.1: ATL Helper: getAssociations.

```
helper def: getReceiveLifelines(): Sequence(UML2!"uml::Lifeline") =
  thisModule.getMessages() -> collect(re | re.receiveEvent.covered).first
      ();
```

Listing 3.2: ATL Helper: getReceiveLifelines.

```
helper def: getSendLifelines(): Sequence(UML2!"uml::Lifeline") =
  thisModule.getMessages() -> collect(se | se.sendEvent.covered).first();
```

Listing 3.3: ATL Helper: getSendLifelines.

For the main transformation behavior, the matched rule `Model` shown in Listing 3.4, as it is called, matches its input pattern of type `UML2!"uml::Model"` within the *from* section. The rule then optionally assigns new values to its attributes in the *to* section. Additionally, the main elements of the class diagram are appended to the UML element `packagedElement` as a sequence of types: lifeline, constraint, association and class.

```
rule Model {
  from
    s: UML2!"uml::Model"
```

```
  to
    t: UML2!"uml::Model" (
      name <- s.name,
      ownedRule <- s.ownedRule,
      packagedElement <- thisModule.getLifelines() -> union(thisModule.
          getConstraints()) -> union(thisModule.getAssociations() ->
          iterate(iter; a: Sequence(UML2!"uml::Association") = Sequence{} |
               a.
          append(thisModule.Association(iter.at(1), iter.at(2)))))
    )
}
```

Listing 3.4: ATL Rule: Model.

In the last line of Listing 3.4, `thisModule.Association(...)` is called as a unique lazy rule, which is executed and always returns the same target element for a given source element [2]. Listing 3.5 processes the lifeline pairs produced by the helper discussed earlier. For each lifeline pair, an association is generated in the *to* section of the rule. In addition to the name assignment, the `ownedEnd` attribute of the association is set for the corresponding class as well. In the *do* section, the target pattern `t` with type association is returned analogous to a normal return value in *Java*.

```
unique lazy rule Association {
  from rcv: UML2!"uml::Lifeline", snd: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Association" (
      name <- rcv.name + '_' + snd.name,
      --   memberEnd <-
      ownedEnd <- Sequence{thisModule.AssociationOwnedEnd(rcv, snd)}
    )
  do {
    t; -- return generated association
  }
}
```

Listing 3.5: ATL Rule: Association.

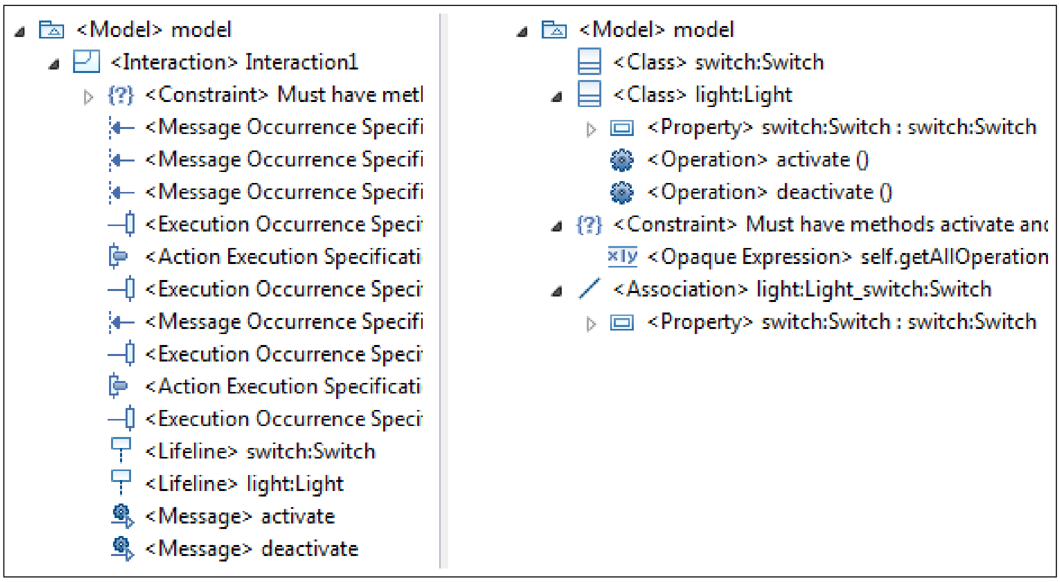
In order to transform lifelines to classes, Listing 3.6 shows the source code for this particular behavior. At first, simple attributes are applied as they are set for the lifeline. In the end, the `ownedAttribute` for the possible association is added.

```
rule Lifeline2Class {
  from
    s: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Class" (s
      name <- s.name,
      visibility <- s.visibility,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
      ownedOperation <- thisModule.getMessages(),
      ownedAttribute <- let assList: Sequence(OclAny) =
          thisModule.getAssociations()
        in
          if assList -> isEmpty() then
            Sequence {}
```

```
     else
      let a: Sequence(OclAny) =
       assList -> select(a | if a -> at(1) = s then
          true
         else
          false
         endif)
      in
       if a -> isEmpty() then
        Sequence {}
       else
        Sequence {}.append(thisModule.
          ClassOwnedAttributeAssociation(a -> flatten() ->
          at(1), a -> flatten() -> at(2)))
       endif
     endif
  )
}
```

Listing 3.6: ATL Rule: Lifeline2Class.

More details of the transformation module can be looked up in Appendix A.1. As the most important rules are declared by now, we will take a look at Figure 3.2 representing the input model on the left and the output model on the right respectively. The generated classes represent the former lifelines. Messages were transformed to operations and are owned by its receiver lifeline represented by a class. The constraint was copied through a normal matched rule, applying the input attributes to the output ones. Concluding with the class association element which was generated due to a message connecting the lifelines `Light` and `Switch`.



- <Model> model
  - <Interaction> Interaction1
    - <Constraint> Must have metl
    - <Message Occurrence Specifi
    - <Message Occurrence Specifi
    - <Message Occurrence Specifi
    - <Execution Occurrence Speci
    - <Action Execution Specificati
    - <Execution Occurrence Speci
    - <Message Occurrence Specifi
    - <Execution Occurrence Speci
    - <Action Execution Specificati
    - <Execution Occurrence Speci
    - <Lifeline> switch:Switch
    - <Lifeline> light:Light
    - <Message> activate
    - <Message> deactivate

- <Model> model
  - <Class> switch:Switch
  - <Class> light:Light
    - <Property> switch:Switch : switch:Switch
    - <Operation> activate ()
    - <Operation> deactivate ()
  - <Constraint> Must have methods activate anc
    - <Opaque Expression> self.getAllOperation
  - <Association> light:Light_switch:Switch
    - <Property> switch:Switch : switch:Switch

Figure 3.2: Light Switch UML Model - Sequence to Class Transformation.

This chapter demonstrated a model-to-model transformation example. For the main part of this thesis we will focus on constraint-driven scenario transformations in the next chapter.

Chapter 4

Implementation and Discussion

As this thesis' major focus lies on the implementation of ATL transformations and OCL constraint generation, this chapter will document 9 selected scenarios in detail. We will not only take a look onto the formal specifications, but also discuss implementation design choices for each scenario within the documentation. In order to overcome the complexity of the UML metamodel, some restrictions to the semantics of UML models had to be made during development. These limitations indeed affect the variety of the used UML models, however, adaptions to support a wider diversity could be done easily and therefore are referenced as future work. Nevertheless, the most interesting and non-trivial constraint-driven scenarios were chosen to showcase the capabilities of the ATL and OCL when checking and fixing (in)consistencies in UML models. In the first section we will go through each of the nine scenarios and conclude the documentation in the second section with the usage documentation of the proposed framework.

4.1 Constraint-driven Scenarios

In the paragraph above, the complex nature of UML models were mentioned as it leads to an indefinite number of consistency scenarios. Besides being standardized, the UML has to be checked for its *syntactic* as well as *semantic consistency*. There is a clear difference between the usual validation of UML models based on syntactic validation (e.g. in Papyrus via the Validate command) and the more specific check, whether the class diagram conforms to its sequence diagram and/or statemachine. In the latter case, we will from here on refer as semantic validation. Many researchers have been working on this subject before, e.g.

in [17], Egyed and Reder created an instant and incremental consistency management framework called *Model/Analyzer*. A catalogue containing a wide variety of design rules for UML models, on which the framework operates upon, is available on the institute website [1]. Basically, this rule set provided a general starting point for the formalization of basic consistency rules used in this thesis.

Obviously, each of the forthcoming scenarios depend on the formal definition (denoted as *Formalization* and written in Prosa) of such design rules and/or similar but mostly more complex ones. In context of the *Transformation*, the formalization has to be rewritten in the ATL. Based on ATL rules it is then decided, whether potential inconsistencies occur and further must be fixed during the transformation process. One easily can imply, that the *Validation* for the corresponding OCL expression is necessary too. As the ATL is built upon the OCL, the validation is relatively straight forward, given the OCL expression for its scenario is defined. The outcome for each of the transformation scenarios will cover constraints in form of OCL expressions and the action on the UML model itself, but only if a constraint was violated. The reason for not fixing certain constraint violations depends on the fact that ambiguous scenarios produce non-deterministic choices where the transformation itself can not be automated anymore. Hence user input would be needed. As all possible fixing scenarios would lead beyond the required effort for this thesis, it is refered as future work. Due to the used EMFTVM supporting in-place transformations, incremental behavior is still applied. In fact, only the model delta, triggered by the transformation rules, is saved on the same model again. Untouched elements are not copied. [3]

The UML models under test were introduced in Section 2.1 already and in case of changes - only for demonstration purposes - will be shown again when necessary. For the readers convenience, the UML abbreviation for any UML element is omitted in the scenario sections when not necessary. In general, when talking about *rules*, *helpers*, *source/target patterns*, *from*, *using*, *to* and *do* sections, the ATL abbreviation is omitted as well. Although a general introduction to ATL as well as a brief transformation example were given in the sections before, the forthcoming ATL transformations are far more complex and may not be that easy to understand in the first place. Hence the first scenario will be explained and documented in much more detail than the following ones.

---

[1] Online at: `http://www.jku.at/sea/content/e139529/e126342/e126449/`.

4.1.1 Message - Operation

For the first scenario and as the heading of this subsection states, we will investigate the relationship between messages (occurring in the UML sequence diagram) and their corresponding operations (occurring in the UML class diagram). In [23], Demuth et al. pointed out some preliminary considerations: when a message does not have its operation represented and the owner class is part of an inheritance hierarchy with at least one superclass, it can not be determined to which class in the inheritance hierarchy the operation should be added. For this scenario we refer to Figure 2.1, where `A` is a subclass of `C`, messages `x` and `z` enforce operations `x()` and `z()` inside the inheritance hierarchy. The inheritance hierarchy starts with class `A` and ends at its topmost superclass `C`. Now that we have described the situation, a formal and general statement can be formalized. To avoid any misinterpretations, the formalization statement below is identical to the name and potential comments in the implementation. The source code can be looked up in Appendix A.2.

Formalization

For each message, its corresponding operation must exist inside the class inheritance hierarchy.

Transformation

We already have learned about the ATL in Section 2.2 and seen a typical ATL transformation example in Chapter 3. Hence, we can spare the basics and immediately focus on the implementation as well as design choices for this particular scenario.

In order to perform the desired transformation for the formalization above, the rule must be in the right context. For a matched rule, the very starting point of the transformation begins with the source pattern. Without any doubt, this should be the message, as it makes sense to build the OCL expression for the message context. But the issue with this approach is that an OCL expression is not permitted to be attached to any message w.r.t. the UML metamodel. A simple bypass for this problem can be implemented as follows: Change the rule context to the class and add helpers to get all messages for the corresponding lifeline. Listing 4.1 and Listing 4.2 both show the helpers, where the `getMessagesByClass` selects all messages for a given class. Another helper `getReceiverLifelineClass` is called to retrieve the class for each message which is then matched to the given class in context.

```
helper def: getReceiverLifelineClass(m: UML2!Message): UML2!Class =
  UML2!Lifeline.allInstancesFrom(’INOUT’) -> select(l | l.coveredBy ->
       select(i | i.
    oclIsTypeOf(UML2!MessageOccurrenceSpecification)) -> exists(e | e =
         m.
    receiveEvent)) -> first().represents.type;
```

Listing 4.1: ATL Helper: getReceiverLifelineClass.

```
helper def: getMessagesByClass(cl: UML2!Class): Sequence(UML2!Message) =
  UML2!Message.allInstancesFrom(’INOUT’) -> select(m | thisModule.
    getReceiverLifelineClass(m) = cl);
```

Listing 4.2: ATL Helper: getMessagesByClass.

Each message will then be processed in a set containing all messages for the lifeline represented by the class in context. On the one hand, this design choice facilitates the handling of multiple constraint violations via concatenation of all message-related OCL expressions. On the other hand, the broader context of the class might rise the complexity of navigational statements inside OCL expressions. But for this scenario, it is of no concern since messages can be retrieved via helpers and then have their names hardcoded within the OCL expression.

Besides retrieving the right set of UML elements via helpers, the main part of the transformation is carried through the matched rule (Listing 4.3 - 4.6). The source pattern is implemented as *from* section and corresponds to the rules context with an optional conditional guard statement `s.oclIsTypeOf(UML2!Class)`. In fact, this condition denotes the rule as *matched* rule, because only elements of type class will match. Without the guard condition, e.g. UML elements of type UML statemachine also would be considered and therefore result in undefined behavior.

```
rule Class {
  from
    s: UML2!Class (
      s.oclIsTypeOf(UML2!Class)
    )
```

Listing 4.3: ATL Rule: from Section.

The *using* section mainly covers simple variable definitions, but more importantly, the declarative expression, where the helpers are called. We already retrieved all messages and further compare those messages with all owned operations for the current class in context. In the formalization section, we exactly defined this scenario, which is either satisfied, when the set of operations is empty, or violated when the set of operations is not empty.

```
 using {
   c01Name: String = 'For the class \'' + s.name + '\', each message must
       be' + ''
     + ' represented by an operation and inside the corresponding class'
        + ''
     + ' hierarchy.';
   c01Expr: String = OclUndefined;
   c01Elements: Sequence(UML2!Message) = OclUndefined;
   newOps: Sequence(UML2!Message) = thisModule.getMessagesByClass(s) ->
     debug('ConcurrentModificationException Fix') -> select(m | not s.
     ownedOperation -> exists(o | o.name = m.name));
 }
```

Listing 4.4: ATL Rule: using Section.

In the *to* section, nothing should happen since the class in context should not be altered every time the matched rule is executed. The imperative *do* section will take care of this behavior.

```
 to
   t: UML2!Class (
     -- keep class properties
   )
```

Listing 4.5: ATL Rule: to Section.

Finally, as the *do* section provides imperative behavior, actions based on the set of operations can be specified. In Listing 4.6, an enclosing for loop embodies the set of operations.

For each operation, the *called* rule `NewOperation` generated as a new operation instance in Listing 4.7. The returned operation is then appended either to its owner, which is the UML model as its whole when there is at least one superclass, or to the class in context, when no generalization exists at all. In the former case, a comment is added to the operation in order to signal the non-deterministic decision due to multiple potential owner classes.

Still, the OCL expression has to be built for all new operations. Beginning at line 24, every message of the class in context is concatenated with its name and compared to the corresponding operation name.

```
1  do {
2     -- add missing operations
3    for (m in newOps) {
4      -- when there is no super class, add operation to class
5      if (not s.allOwnedElements() -> exists(g | g.
6        oclIsTypeOf(UML2!Generalization))) {
7       thisModule.NewOperation(m.name, '', s);
8      }
9      -- otherwise add operation to model, in case it does not exist yet
10     else if (UML2!Operation -> allInstancesFrom('INOUT') -> select(o |
```

```
      o.
      owner = OclUndefined and o.ownedComment -> exists(oc | oc.body =
      c01Name)) -> isEmpty()) {
     thisModule.NewOperation(m.name, c01Name, OclUndefined);
    }
   } -- get all messages for constraint expression
   c01Elements <- thisModule.getMessagesByClass(s);

  -- for each operation, build constraint
  if (c01Elements -> size() > 0) {
   c01Expr <- 'self.inheritedMember->select(oclIsTypeOf(Operation))->
        union(self.ownedOperation)->exists(name=\'' + c01Elements.first()
        .
     name + '\')';

   c01Elements <- c01Elements -> subSequence(2, c01Elements -> size());
   for (o in c01Elements) {
    c01Expr <- c01Expr.concat(' and self.' +
      'inheritedMember->select(oclIsTypeOf(Operation))->union(self.
         ownedOperation)->exists(name=\'' + o.name + '\')');
   } -- add constraint to class
    if (not s.allOwnedElements() -> select(c | c.
     oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c01Name) and
     s.oclIsTypeOf(UML2!Class)) {
    thisModule.NewOwnedRule(s, c01Name, c01Expr, 'OCL');
   }
  }
 }
```

Listing 4.6: ATL Rule: do Section.

```
-- new operation constructor alternative
rule NewOperation (oStr: String, cStr: String, owner: OclAny){
 using {
  o: UML2!Operation = UML2!Operation.newInstanceIn('INOUT');
  c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
 }
 do{
  c.body <- cStr;
  o.name <- oStr -> debug('ADD operation');
  if (owner <> OclUndefined) o.class <- owner;
  o; -- return operation
 }
}
```

Listing 4.7: ATL Called Rule: NewOperation.

The OCL expression is the second new instance which should be generated within the transformation. With respect to the UML metamodel, a constraint with the concatenated OCL expression enclosed as opaque expression is generated in Listing 4.8. The returned constraint is then appended to its owner, which is represented by the class in context.

```
-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
   String) {
 using {
  c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
  oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('INOUT
    ');
 }
 do {
  oe.language <- oe.language -> append(l);
  oe.body <- oe.body -> append(exp);
  c.name <- ruleName -> debug('ADD ownedRule');
  c.constrainedElement <- c.constrainedElement -> append(owner);
```

```
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}
```

Listing 4.8: ATL Called Rule: NewOwnedRule.

When the transformation is performed onto the UML Inheritance model (Figure 2.1), the console output lists the new instances generated as shown in Listing 4.9. One can observe, the two messages `x` and `z` were added to the UML model. For both a comment signals the absence of UML ownership, because the owner class is in fact a subclass.

```
ADD operation: ’x’
ADD operation: ’z’
ADD ownedRule: ’For the class ’A’, each message must be represented by an
     operation and inside the corresponding class hierarchy.’
TEST: model transformation successful ...
```

Listing 4.9: Transformation Console Output: Message - Operation.

The transformed model shows both new operations appended outside the scope of the actual UML model (as seen in the XML representation in Listing 4.10).

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI
    /20110701" xmlns:uml="http://www.eclipse.org/uml2/4.0.0/UML">
  <uml:Model xmi:id="_OREfYOewEeKJ0egLp7Bx_w" name="model">
  ...
  </uml:Model>
  <uml:Operation xmi:id="_zLCOIPkyEeK3noywWhB5XA" name="x"/>
  <uml:Comment xmi:id="_zLCOIfkyEeK3noywWhB5XA">
    <body>For the class ’A’, each message must be represented by an
        operation and inside the corresponding class hierarchy.</body>
  </uml:Comment>
  <uml:Operation xmi:id="_zLIUwPkyEeK3noywWhB5XA" name="z"/>
  <uml:Comment xmi:id="_zLIUwfkyEeK3noywWhB5XA">
    <body>For the class ’A’, each message must be represented by an
        operation and inside the corresponding class hierarchy.</body>
  </uml:Comment>
</xmi:XMI>
```

Listing 4.10: XML Output Model: Message - Operation.

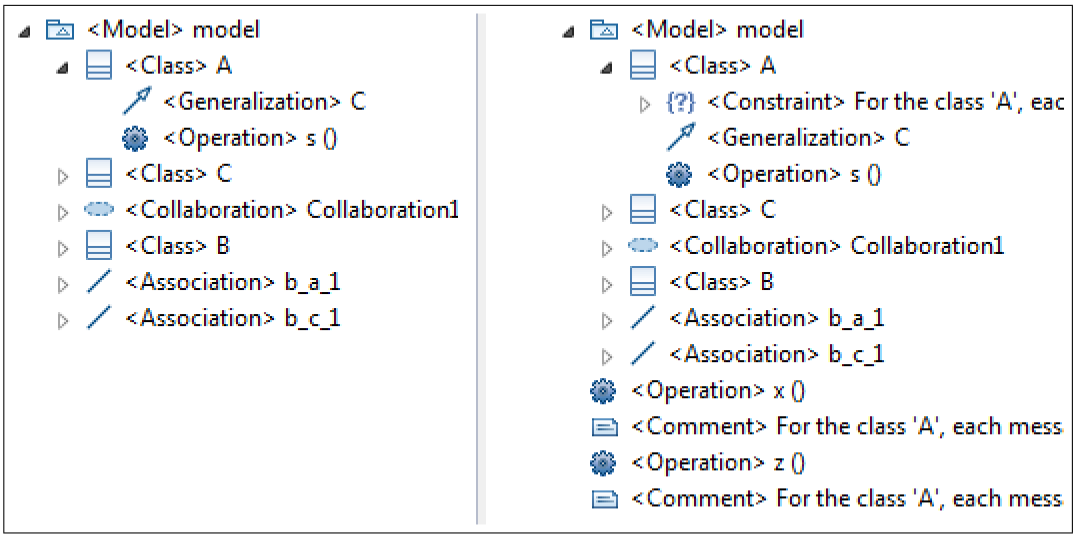
Validation

Concerning the OCL expression generated, this small section will show the correct outcome of both the transformation and the OCL validation. Before the actual validation is done, we will take a brief look onto the generated OCL expression in Listing 4.11. Similar to the matched rule in the transformation section, the OCL expression does need its context specified as well. Identical to the transformation, the context is UML class. For each class, which is represented by the keyword `self`, all operations have to exist for its message representations. Stated with the `union` keyword, an operation is either inherited or owned by the class itself.

```
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='x') and self.inheritedMember->select(
    oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(name='z')
```

Listing 4.11: OCL Expression: Message - Operation.

For the purpose of validation, the UML Model Editor OCL Console (Eclipse plug-in) is used. The left model of Figure 4.1 first demonstrates the absence of both operations. The validation for the right model returns `false`, which means the constraint is not satisfied. Both operations are not in the context of class `A` (Listing 4.12).



```
◢ <Model> model
    ◢ <Class> A
          <Generalization> C
          <Operation> s ()
    ▷ <Class> C
    ▷ <Collaboration> Collaboration1
    ▷ <Class> B
    ▷ <Association> b_a_1
    ▷ <Association> b_c_1
```

```
◢ <Model> model
    ◢ <Class> A
        ▷ {?} <Constraint> For the class 'A', eac
          <Generalization> C
          <Operation> s ()
    ▷ <Class> C
    ▷ <Collaboration> Collaboration1
    ▷ <Class> B
    ▷ <Association> b_a_1
    ▷ <Association> b_c_1
  <Operation> x ()
  <Comment> For the class 'A', each mess
  <Operation> z ()
  <Comment> For the class 'A', each mess
```

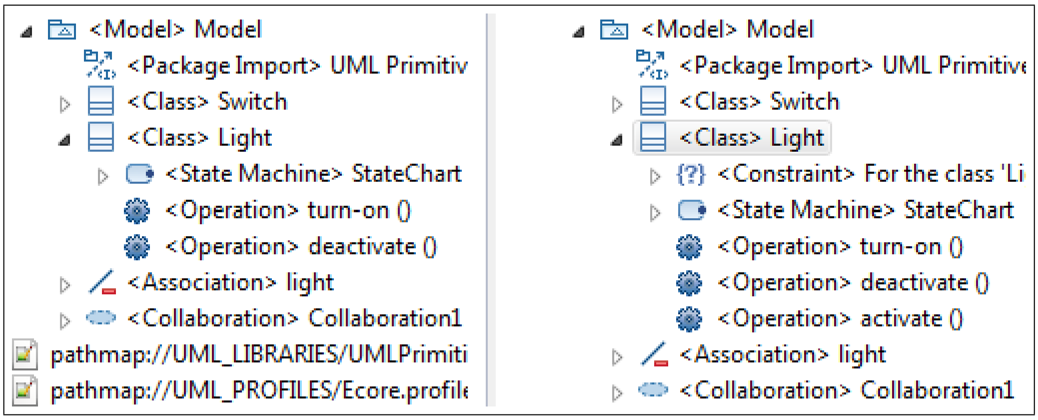
Figure 4.1: Inheritance UML Model Editor Validation.

```
Evaluating:
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='x') and self.inheritedMember->select(
    oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(name='z')
Results:
false
```

Listing 4.12: OCL Validation : Message - Operation.

Since the output model could not satisfy the constraint, another test case is conducted using the Light Switch UML Model shown in Figure 2.2. This model does not contain any generalizations, hence the missing `activate()` operation should be added to its owner class `Light` correctly. Doing so, the OCL expression evaluates to `true`. Again, the left side represents the input model which is then transformed to its right representation, the output model.

Just a quick reminder, when talking of in- and output models: in fact, there is only one 'INOUT' model, as the input model is simply transformed during refining mode. For



<Model> Model
- <Package Import> UML Primitiv
- <Class> Switch
- <Class> Light
  - <State Machine> StateChart
  - <Operation> turn-on ()
  - <Operation> deactivate ()
- <Association> light
- <Collaboration> Collaboration1

pathmap://UML_LIBRARIES/UMLPrimiti
pathmap://UML_PROFILES/Ecore.profil

<Model> Model
- <Package Import> UML Primitive
- <Class> Switch
- <Class> Light
  - <Constraint> For the class 'Li
  - <State Machine> StateChart
  - <Operation> turn-on ()
  - <Operation> deactivate ()
  - <Operation> activate ()
- <Association> light
- <Collaboration> Collaboration1

Figure 4.2: Light Switch UML Model Editor Validation.

testing purposes and to preserve the input models in its initial state, the output model is saved as another file. In Listing 4.13 one can observe the evaluation was actually successful, because `activate` was added to class `Light`.

```
Evaluating:
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='activate') and self.inheritedMember->
    select(oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(
    name='deactivate')
Results:
true
```

Listing 4.13: OCL Validation: Message - Operation.

4.1.2 Lifeline - Class

This scenario covers the lifeline - class connection. Based on the UML navigation `represents.type`, each lifeline is linked to its type via the lifeline instance property and then again via the property type, which designates the class in the class diagram. In other words, a property in the sequence diagram connects both lifeline and class. In general, a sequence diagram provides a subset of the information of the corresponding class diagram. Hence it is indeed possible that lifelines are missing, or even further: no sequence diagram exists at all. The other way around, it is assumed: for each lifeline, the corresponding class exists. But the property and the corresponding class of the lifeline may or may not exist. This leads to another issue, where the lifeline name must start with a capital letter. In case of violation, the first character will be capitalized automatically so that the lifeline property can be defined with the lowercase version of the name.

As this scenario does not require a specific UML model at all, an arbitrary model, e.g. Inheritance UML model (Figure 2.1) is chosen for validation. In order to provide a meaningful test case, an additional lifeline with a unique name is added manually. The detailed implementation can be looked up in Appendix A.3.

Formalization

For each lifeline, a corresponding class must exist.

Transformation

On the contrast to the first scenario, where a class may own multiple operations, a lifeline is always connected to exactly one class and denoted as one-to-one relationship according to the UML metamodel. Based on this fact, we had to change the context of the rule class, whereas in this scenario, we always will retrieve exactly one or no class at all. This comes quite handy when appending OCL expressions for the lifeline. The OCL expression is therefore appended to the class representing the existing or newly created lifeline. For this reason, the context for the ATL rule can remain as lifeline and the generated constraint is appended to the lifeline class. We already have mentioned the link between a lifeline and its class representation. In Listing 4.14, three different cases are distinguished. Both properties through has to be checked and in return only the class for the full link can be evaluated.

```
helper def: getLifelineClass(l: UML2!Lifeline): UML2!Class =
  if (l.represents = OclUndefined) then
    OclUndefined
  else
    if (l.represents.type = OclUndefined) then
      OclUndefined
    else
      l.represents.type
    endif
  endif;
```

Listing 4.14: ATL Helper: getLifelineClass.

The matched rule's *from* section is guarded with its UML type, in this case, the lifeline. In the *using* section, a simple initial capitalization for the lifeline naming convention (capital first letter) is expressed in the declarative statement shown in Listing 4.15. It simply capitalizes the first letter and concatenates the rest of the name starting at character two. The new string is then assigned in the *to* section of the transformation by assigning the new string to the target pattern attribute *name*.

```
validLlName: String = s.name.at(1) -> toUpper() + s.name.substring(2);
```

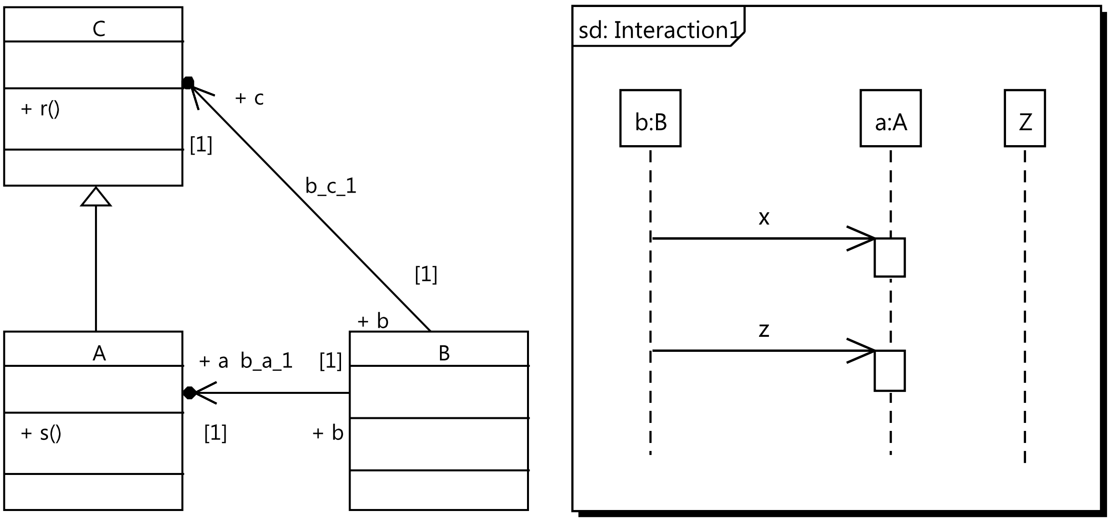
Listing 4.15: ATL Expression: toUpper.

The actual behavior of the transformation is described in the *do* section. In fact, depending on the helpers' return value, a class may or may not exist. In the former case, only the constraint is generated and appended to the lifeline class. For the latter, it must be distinguished, whether the class exists but is not linked to the lifeline, or the class is just missing. And this brings us to the last possibility that the connective property between lifeline and class might not exist. If so, a new property instance is generated as well.

Called rules for the UML element instance generation - for a class, property and constraint - are similar to the ones in the scenario before and will not be listed again. We rather will take a look at the console output shown in Listing 4.16. For this particular test case, an arbitrary lifeline without any property was added to the sequence diagram (Figure 4.3). As a result, a property as well as the class were generated. Each lifeline class is then extended with its corresponding constraint.

```
ADD ownedRule: 'For each lifeline, a corresponding class must exist.'
ADD ownedRule: 'For each lifeline, a corresponding class must exist.'
ADD class: d391410:UML2!Class
ADD property: 704a8a11:UML2!Property
ADD ownedRule: 'For each lifeline, a corresponding class must exist.'
TEST: model transformation successful ...
```

Listing 4.16: Transformation Console Output: Lifeline - Class.



(a) Class Diagram.

(b) Sequence Diagram.

Figure 4.3: Inheritance UML Model Example.

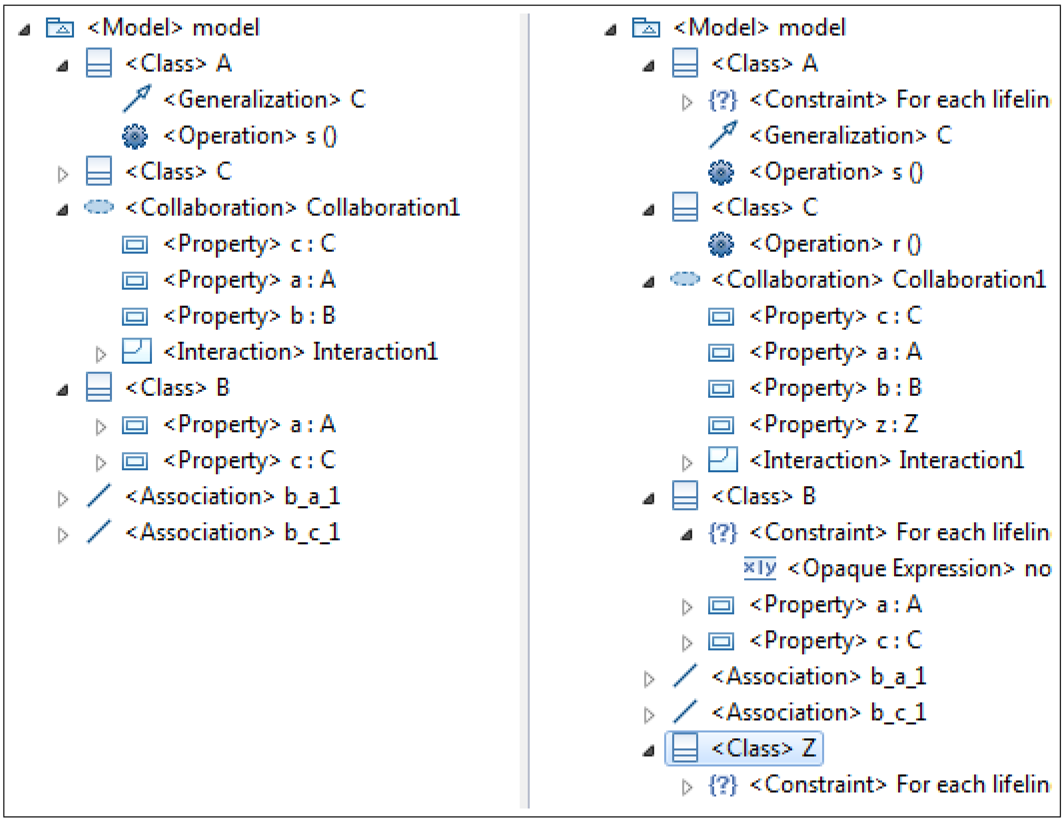
Validation

In Listing 4.17 the OCL expression is triggered for the lifeline class in context. In particular, `Z` is selected out of all lifelines. For the single lifeline, the property as well as the type (which corresponds to class `Z`) must not be undefined.

```
Lifeline.allInstances()->select(name = 'Z').represents.type->notEmpty()
```

Listing 4.17: OCL Expression: Lifeline - Class.

Figure 4.4 shows the compared models before (left) and after the transformation executed (right). Validating the constraint for class `Z` results in `true` (shown in Listing 4.18), since the property `z` and its class `Z` were generated and added to the model on the right.



- <Model> model
  - <Class> A
    - <Generalization> C
    - <Operation> s ()
  - <Class> C
  - <Collaboration> Collaboration1
    - <Property> c : C
    - <Property> a : A
    - <Property> b : B
    - <Interaction> Interaction1
  - <Class> B
    - <Property> a : A
    - <Property> c : C
  - <Association> b_a_1
  - <Association> b_c_1

- <Model> model
  - <Class> A
    - <Constraint> For each lifelin
    - <Generalization> C
    - <Operation> s ()
  - <Class> C
    - <Operation> r ()
  - <Collaboration> Collaboration1
    - <Property> c : C
    - <Property> a : A
    - <Property> b : B
    - <Property> z : Z
    - <Interaction> Interaction1
  - <Class> B
    - <Constraint> For each lifelin
      - <Opaque Expression> no
    - <Property> a : A
    - <Property> c : C
  - <Association> b_a_1
  - <Association> b_c_1
  - <Class> Z
    - <Constraint> For each lifelin

Figure 4.4: Inheritance UML Model Editor Validation.

```
Evaluating:
Lifeline.allInstances()->select(name = 'Z').represents.type->notEmpty()
Results:
true
```

Listing 4.18: OCL Validation: Message - Operation.

4.1.3 Transition - Operation

Just like the first scenario, where each message is represented by its operation, each transition in a statemachine must be represented by its operation too. As the owner of statemachine must be of type class, the class is chosen for the UML element in context considering the transformation rule, OCL expression and constraint generation. Additionally, a statemachine is a subset of the class diagram (analogous to the sequence diagram). That is why the given scenario must not be fulfilled in the opposite way. Indeed, one has the freedom to further constrain and adapt the UML metamodel, however for the purpose of the transformation demonstration, we omit this.

This scenario does require a new UML model under test which at least contains one statemachine. Since the VOD UML model (Figure 2.3) does not violate the constraint described, the VOD AR UML model (Figure 2.4 and Figure 2.5) is chosen for validation. The complete implementation can be looked up in Appendix A.4.

Formalization

For each transition, a corresponding operation must exist.

Transformation

Due the rule in context being of type class, a helper comes in handy to provide every transition for a class. In Listing 4.19, such helper matches all transitions to their owner. More precisely, three owner attributes are navigated starting at transition attribute: the first representing the `region` element, the second the `ownedBehavior` and the third the class itself. The OCL expression listed later in this section, obviously will be built from all transitions and appended to the class.

```
helper def: getTransitionsByClass(cl: UML2!Class): Sequence(UML2!
    Transitions) =
UML2!Transition.allInstancesFrom('INOUT') -> select(t | t.owner.owner.
    owner = cl);
```

Listing 4.19: ATL Helper: getTransitionsByClass.

With the required helpers implemented, we will then filter the missing transitions by a given class. Therefore we will make use of `getTransitionsByClass`, as shown above, and select the transitions not having its operations defined.

```
newOps: Sequence(UML2!Transition) = thisModule.getTransitionsByClass(s)
    -> select(tr | not s.ownedOperation -> exists(o | o.name = tr.name));
```

Listing 4.20: ATL Expression: newOps.

The main matched rule is analogous to the Message - Operation Scenario in Subection 4.1.1 and for that reason not listed anymore. The same applies to the called rules for generating the operations and comments. In this scenario, the statemachine does own a transition which the owning class has not specified. Hence the console output in Listing 4.21 shows the fix as expected: the operation `init()` is added to class `Streamer` directly. Again, the class generalization attribute is checked whether an inheritance hierarchy exists or not. As `Streamer` is not a subclass, the fix can be carried out.

```
ADD operation: 'init'
ADD ownedRule: 'For each transition, a corresponding operation must exist
    .'
TEST: model transformation successful ...
```

Listing 4.21: Transformation Console Output: Transition - Operation.

Before the validation, we want to discuss the choice for class as context. In later scenarios, constraints are appended to statemachines, because statemachines do permit constraint ownership just like a class.

If we would choose the statemachine as context, multiple statemachines owned by the same class would produce multiple constraints. Therefore, multiple OCL interpretations would potentially prolong the validation. However, for small models under test it would not make any reasonable difference in performance at all.

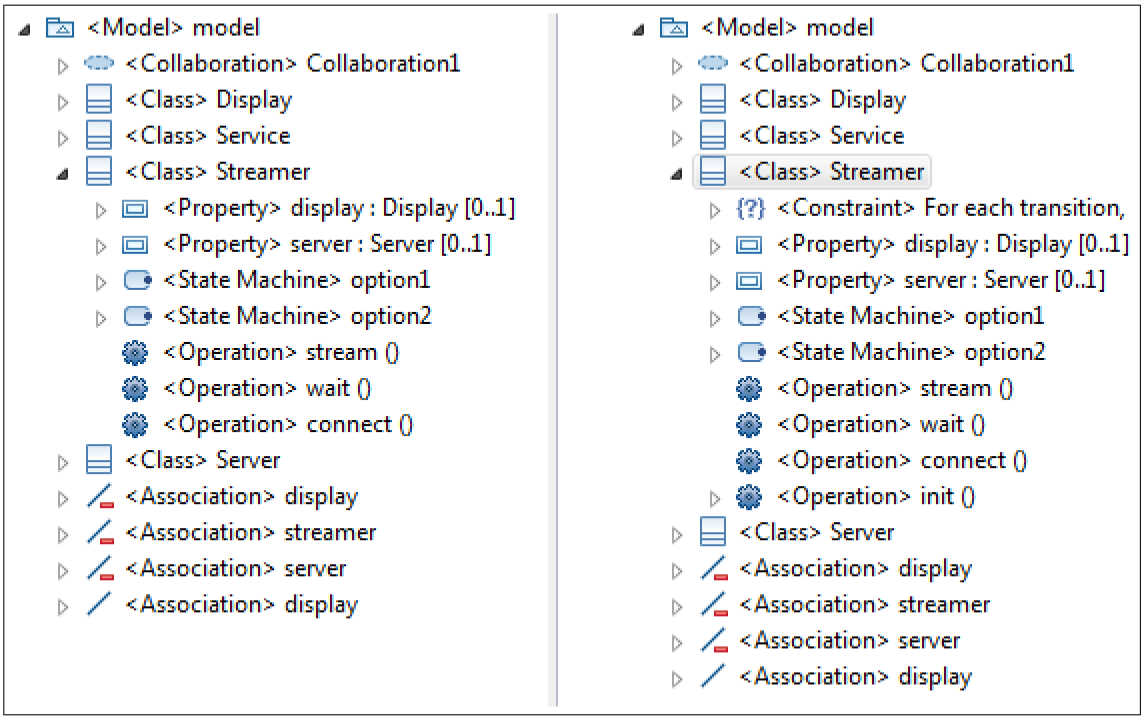
Validation

The generated OCL expression combines inherited operations as well as the operations owned by the class. The set of all occurring transitions will then be matched to the output model on the right hand side of Figure 4.5. As *init()* was added through the process of the transformation, the constraint is fulfilled (Listing 4.22).

```
Evaluating:
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='stream') and self.inheritedMember->
    select(oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(
    name='wait') and self.inheritedMember->select(oclIsTypeOf(Operation))
    ->union(self.ownedOperation)->exists(name='stream') and self.
    inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='wait') and self.inheritedMember->select
    (oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(name='
    connect') and self.inheritedMember->select(oclIsTypeOf(Operation))->
    union(self.ownedOperation)->exists(name='init') and self.
    inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='wait') and self.inheritedMember->select
    (oclIsTypeOf(Operation))->union(self.ownedOperation)->exists(name='
    stream') and self.inheritedMember->select(oclIsTypeOf(Operation))->
    union(self.ownedOperation)->exists(name='connect')
Results:
true
```

Listing 4.22: OCL Validation: Transition - Operation.

The unification of `inheritedMember` and `ownedOperation` does look rather complex and one might wonder why there is no combined statement for retrieving all operations in



- <Model> model
  - <Collaboration> Collaboration1
  - <Class> Display
  - <Class> Service
  - <Class> Streamer
    - <Property> display : Display [0..1]
    - <Property> server : Server [0..1]
    - <State Machine> option1
    - <State Machine> option2
    - <Operation> stream ()
    - <Operation> wait ()
    - <Operation> connect ()
  - <Class> Server
  - <Association> display
  - <Association> streamer
  - <Association> server
  - <Association> display

- <Model> model
  - <Collaboration> Collaboration1
  - <Class> Display
  - <Class> Service
  - <Class> Streamer
    - <Constraint> For each transition,
    - <Property> display : Display [0..1]
    - <Property> server : Server [0..1]
    - <State Machine> option1
    - <State Machine> option2
    - <Operation> stream ()
    - <Operation> wait ()
    - <Operation> connect ()
    - <Operation> init ()
  - <Class> Server
  - <Association> display
  - <Association> streamer
  - <Association> server
  - <Association> display

Figure 4.5: VOD AR UML Model Editor Validation.

an inheritance hierarchy. Actually, UML would support the desire of having one function retrieving both at the same time via the function `'class'.getAllOperations()`. But neither the Xtext OCL interpretation console nor the OCL validation functionality of Papyrus does support the interpretation yet. Hence, the longer version of the expression is used.

4.1.4 Message Sequence - Transition Sequence

Having discussed three standard scenarios, this one aims at the more unusual characteristics of UML models. In the past sections we stated that the sequence diagram as well as the statemachine describe a subset of the information of the class diagram. Nevertheless, messages and transitions share some aspects regarding the class diagram representation. Both elements must be represented as operation and generally speaking share the same execution behavior. A very important characteristic of sequence diagrams and statemachines is the fact that they describe operations in a timed order. Whereas the sequence diagram presents interactions between the lifeline classes, the statemachine does this in an analogous way but is restricted to a class. Since sequence diagrams having the classes represented by lifelines, we can compare the order of messages to a possible path in the statemachine. The lifeline and the statemachine both are owned by a class, thus the class is chosen for the context. This does represent the *Least Common Multiple (LCM)* for lifeline and statemachine.

Due to a class owning multiple sequence diagrams and statemachines, some restrictions were made during development. Only one sequence diagram for the UML model is allowed, but still multiple statemachines per class are considered. The UML model under test will be the VOD AR model (Figure 2.4 and Figure 2.5), as it owns a sequence diagram and two different statemachines for class `Streamer`. The full implementation is available at Appendix A.5.

Formalization

Sequence of messages must match sequence of transitions.

Transformation

We discussed the transformation context in the paragraph above. The class in context is applied for constraint attachment as well as the OCL expression. Following the structure of former scenarios, helpers are introduced first:

Thus, the context being class, messages and transitions are navigated via the helpers below. Those were mentioned earlier and as a result will not be listed once more.

- `getMessagesByClass`
- `getReceiverLifelineClass`
- `getTransitionByClass`

A more interesting helper is shown in Listing 4.23. Given the fact that messages are ordered already but transitions are not, the possible transition paths through its model have to be extracted and saved as an OCL collection type. For each statemachine, `reorderTransitions` starts with the initial pseudostate and recursively walks through all possible paths (using the `iterate` function) analogous to a *Depth-First Search (DFS)* algorithm. Since the later traversal which is needed for validation might abort in the middle of a transition path, another path could still fulfill the correct message order and that is why the DFS is needed. The returning sequence of transitions then represents the reordered list of transitions starting at the pseudostate and ending if a state was already reached through any transition.

```
helper def: reorderTransitions(st: UML2!Vertex, sm: UML2!StateMachine, l:
    Sequence(UML2!Transition), visited: Sequence(UML2!Vertex)):
    Sequence(UML2!Transition) =
  if visited -> exists(e | e = st) then
    l -> append(UML2!Transition.allInstancesFrom('INOUT') -> select(t2 |
         t2.owner.
        owner = sm and t2.source = st))
  else
    -- append transition to list and recursively call function for target
         state
    UML2!Transition.allInstancesFrom('INOUT') -> select(t1 | t1.owner.
         owner = sm and
        t1.source = st) -- for each source
       -> iterate(i; init: OclAny = OclUndefined | -- call recursively
      thisModule.reorderTransitions(i.target, sm, (l -> append(UML2!
           Transition.
         allInstancesFrom('INOUT') -> select(t2 | t2.owner.owner = sm and
              t2.
         source = st))), visited -> append(i.source)))
  endif;
```

Listing 4.23: ATL Helper: reorderTransitions.

As messages and transitions both are in the correct order - messages are ordered already w.r.t. to the sequence diagram encapsulated in the UML model, the sequence of messages for a specific lifeline are traversed in listing 4.24. Each message is accessed by its order index `i`, for which the corresponding transition is looked up in a sequence of transitions without gaps (cyclic paths are possible though). Given the current state (represented by the UML element called `vertex`) of the traversal, the current message name has to exist in the subset of transitions. We call it a subset of transitions, because for each state multiple transitions might be possible. This is analogous to the graph built according to DFS characteristics. Normally, if no transition conforms to the current message, the constraint is violated. Additionally, a successful match of both sequences is specified so that the sequence of transitions might be a subset of the messages respectively. Therefore, as we will see in the later validation, it is possible that the matching sequence begins with the first message but not necessarily with the first outgoing transition of the pseudostate in the statemachine. Further, the matched sequence must be present in all statemachine instances for the UML models class. The helper either returns `0` if the constraint is satisfied and the position of the current message if violated.

```
helper def: traverse(st: UML2!Vertex, i: Integer, t: Sequence(UML2!
     Transition), msgs:
    Sequence(UML2!Messages), tnsns: Sequence(UML2!Transition)): Integer =
  if msgs.at(i) = msgs -> last() and t -> exists(tr | tr.name = msgs.at(i
       ).name) then
    0
  else
    if not t -> exists(tr | tr.name = msgs.at(i).name) then
      if t = tnsns -> at(1) then
        thisModule.traverse(t -> select(tr | tr.source = st) -> at(1).
             target, i,
            tnsns -> select(tri | t -> select(tr | tr.source = st) -> at(1).
```

```
     target = tri -> at(1).source) -> flatten(), msgs, tnsns)
  else
   i
  endif
 else
  thisModule.traverse(t -> select(tr | tr.name = msgs.at(i).name) ->
     at(1).
    target, (i + 1), tnsns -> select(tri | tri -> exists(e | e.source
       =
    (t -> select(tr | tr.name = msgs.at(i).name) -> at(1).target)))
       ->
    flatten(), msgs, tnsns)
 endif
endif;
```

Listing 4.24: ATL Helper: traverse.

Both recursive helpers might be tricky to understand in the first place, hence it is encouraged to look at the whole implementation in the appendix. A transformation executed onto the VOD AR UML model produces the following output shown in Listing 4.25. As the OCL expression for this scenario exceeds any expectations of this thesis, it is omitted. So for each statemachine, the informal constraint is added in Prosa.

```
ADD ownedRule: 'Sequence of messages must match sequence of transitions.'
ADD ownedRule: 'Sequence of messages must match sequence of transitions.'
CONSTRAINT VIOLATED: 82d195a:UML2!Comment
TEST: model transformation successful ...
```

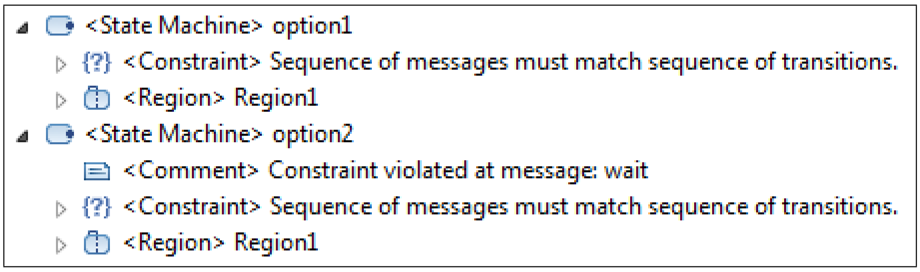
Listing 4.25: Transformation Console Output: Message Sequence - Transition Sequence.

Validation

Listing 4.26 shows the XML snippet and Figure 4.6 the UML model editor view of the transformation output model. The constraint violation happens at the second statemachine for class `Streamer`. This is, because the sequence of messages: `connect, wait, stream` is not applicable for the sequence of transitions: `connect, stream, ...` and aborts at message `wait`.

```
<ownedBehavior xmi:type="uml:StateMachine" xmi:id="
    _qdtI8OcbEeKDavIEctcn2Q" name="option2">
     <ownedComment xmi:id="_r29QYvxPEeKzja5bQDYX_g">
       <body>Constraint violated at message: wait</body>
     </ownedComment>
     <ownedRule xmi:id="_r29QYPxPEeKzja5bQDYX_g" name="Sequence of
         messages must match sequence of transitions." constrainedElement=
         "_qdtI8OcbEeKDavIEctcn2Q">
       <specification xmi:type="uml:OpaqueExpression" xmi:id="
           _r29QYfxPEeKzja5bQDYX_g">
         <language>OCL</language>
         <body></body>
       </specification>
     </ownedRule>
```

Listing 4.26: XML Output Model: Message - Operation.



<State Machine> option1
<Constraint> Sequence of messages must match sequence of transitions.
<Region> Region1
<State Machine> option2
<Comment> Constraint violated at message: wait
<Constraint> Sequence of messages must match sequence of transitions.
<Region> Region1

Figure 4.6: Statemachine Constraint Violation.

4.1.5 Message - Association

Having discussed scenarios for sequence and class diagrams already, one common characteristic was not mentioned yet. As messages do have an anchorpoint for its owner as well as the sending lifeline, each message obviously connects two lifelines. As a matter of fact, both lifelines are represented by its classes and therefore an association must exist. Given the lifelines connected by one or more messages, associations for the corresponding classes can be determined. Because the OCL expression possibly gets more complicated than usual, we will not concatenate them for multiple associations within one class. Preferably, an *endpoint* rule should take care of appending multiple constraints during the end of the transformation. But for now, we do not go into detail and postpone the clarification of this issue to the later stages of this scenario discussion. The context being more important for initial considerations, the aspects for adding constraints and building associations is taken into account. Since associations specify its owner and therefore imply a direction, the messages owner is a lifeline and further represented as class. Hence, specifying class as context is appropriate for this scenario.

Reder and Egyed discussed this design rule in [24] and formalized the OCL expression, but with context being the message. Based on their initial OCL expression, it was the task to further investigate boundary conditions and as a result define actions to fix them. In Chapter 3, we have seen how to build UML associations in general, but another issue arises when the association already exists in the opposite direction. According to the UML metamodel, the `navigable` attribute represents the direction of the association. For this scenario, the VOD AR UML model's (Figure 2.4 and Figure 2.5) association named `display` was turned around. The complete implementation can be looked up in Appendix A.6.

Formalization

A message between two lifelines guarantees an association between the two corresponding classes.

Transformation

Analogous to the first scenario, `getReceiverLifelineClass` (see Listing 4.1) and `getMessagesByClass` (see Listing 4.2) are reused. In addition, the following helper `getMessageLifelineBySendEvent` (see Listing 4.27) returns the sender lifeline for a given `MessageOccurrenceSpecification (MOS)` [12].

```
helper def: getMessageLifelineBySendEvent(snd: UML2!
  MessageOccurrenceSpecification): Sequence(UML2!Lifeline) =
 UML2!Lifeline.allInstancesFrom('INOUT')->select(ll | ll.coveredBy->
   exists(os | os = snd));
```

Listing 4.27: ATL Helper: getMessageLifelineBySendEvent.

In the next snippet, presented in Listing 4.28, the three emerging cases are processed. Basically, if an association is missing, it very well might exist in the opposite direction. If so, the `navigableOwnedEnd` attribute is set to `true`.

```
...
for (snd in sndClass) {
   -- if asso does not exist for snd class, create it
   if (not snd.ownedAttribute->exists(a | a.type = rcvClass)) {
    -- if asso exists in the opposite direction
    if (rcvClass.ownedAttribute->exists(a | a.type = snd)) {
     assoNav <- UML2!Association.allInstancesFrom('INOUT')->select(a |
        a = rcvClass.ownedAttribute->select(a | a.type = snd)->at(1).
       association)->at(1);
     assoNav.navigableOwnedEnd <- assoNav.ownedElement->debug('EDIT
       navigable <- true');
    } else {
     asso <- thisModule.ClassOwnedAttributeAssociation(rcvClass, snd,
       c05Name);
     snd.ownedAttribute <- snd.ownedAttribute->append(asso);
    }
   }
  }
...
```

Listing 4.28: ATL Rule: do Section.

Since association generation was part of the motivating example discussed earlier, it is omitted and simply refer to Chapter 3. Nevertheless, the called rules required are used again and identified as the following:

- `ClassOwnedAttributeAssociation`
- `Association`

- `AssociationOwnedEnd`
- `LiteralInteger`
- `LiteralUnlimitedNatural`

In order to append multiple constraints to the UML element `ownedRule`, the attachment must be executed in the very end of the transformation process. Because the attachment (during the *do* section) would overwrite the last appended element when the same UML model is being attached over and over again. In other words, we can not alter any UML element during transformation if it was altered during the same transformation earlier. This is why we take advantage of an *endpoint* rule for this special case. This rule will be executed right before the end of the transformation. To store the generated constraints during transformation, the global helper sequence `classCons` is used for this purpose.

```
endpoint rule AppendMultipleConstraints () {
  do {
    for (c in thisModule.classCons) {
      c.constrainedElement->at(1).ownedRule <- c.constrainedElement->at(1)
          .ownedRule->append(c);
    }
  }
}
```

Listing 4.29: ATL Endpoint Rule: AppendMultipleConstraints.

Completing the transformation as well as analyzing the console output in Listing 4.30, the `navigableOwnedEnd` attribute is modified and constraints are added to every association.

```
EDIT navigable <- true: Sequence{service:UML2!Property}
ADD ownedRule: 'A message select between two lifelines guarantees an
    association between the two corresponding classes.'
ADD ownedRule: 'A message connect between two lifelines guarantees an
    association between the two corresponding classes.'
ADD ownedRule: 'A message wait between two lifelines guarantees an
    association between the two corresponding classes.'
ADD ownedRule: 'A message stream between two lifelines guarantees an
    association between the two corresponding classes.'
TEST: model transformation successful ...
```

Listing 4.30: Transformation Console Output: Message - Association.

Validation

Similar to [24], the OCL expression extracts the sender as well as the receiver lifeline (in Listing 4.31). The sender class should then own an attribute which is not null, and the type of the attribute must be the class of the receiver class.

```
let l1: Lifeline = Message.allInstances()->select(name = select ).
    receiveEvent.covered.asSequence().at(1) in
let l2: Lifeline = Message.allInstances()->select(name = select ).
    sendEvent.covered.asSequence().at(1) in
let a: Sequence(Property) = l2.represents.type.ownedAttribute in not a =
    null and a.type = l1.represents.type
```

Listing 4.31: OCL Expression: Message - Association.

The fix for an association existing in the opposite direction is then shown in Listing 4.32, where `navigableOwnedEnd` is assigned to the `service` attribute for the association `display`.

```
  <packagedElement xmi:type="uml:Association" xmi:id="_0O8fYO1DEeKsaIbU-
      otI2g" name="display" memberEnd="_0O8fYe1DEeKsaIbU-otI2g
      _0OxgQO1DEeKsaIbU-otI2g" navigableOwnedEnd="_0O8fYe1DEeKsaIbU-otI2g"
      >
    <ownedEnd xmi:id="_0O8fYe1DEeKsaIbU-otI2g" name="service" type="
        _J4teoOcZEeKDavIEctcn2Q" association="_0O8fYO1DEeKsaIbU-otI2g">
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_0O8fYu1DEeKsaIbU-
          otI2g" value="1"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_0O8fY-1
          DEeKsaIbU-otI2g" value="1"/>
    </ownedEnd>
  </packagedElement>
```

Listing 4.32: XML Output Model: Message - Association.

For the third let statement, the OCL interpreter actually fails evaluating `l2.represents`. Analyzing the evaluation of OCL, `l1` and `l2` both evaluate correctly and conform to the type UML lifeline. For some reason, the nested usage and the continued navigation through `represents` fails. A correct validation of this expression is shown in Listing 4.33.

```
Evaluating:
let l1: Lifeline = Message.allInstances()->select(name = 'select' ).
    receiveEvent.oclAsType(MessageOccurrenceSpecification).covered in
let l2: Lifeline = Message.allInstances()->select(name = 'select' ).
    sendEvent.oclAsType(MessageOccurrenceSpecification).covered in
let a: Property = l2.represents.type.ownedAttribute in
not a = null and a.type = l1.represents.type
Results:
true
```

Listing 4.33: OCL Validation: Message - Association.

Before we continue to examine the rest of the scenarios, it is worth mentioning that the forthcoming scenarios will not exaggerate in contrast to the ones discussed previously. For this reason, we will focus on the practical validation part and keep Listings rare.

4.1.6 Statemachine - Class

A more general scenario is based on the ownership of statemachines. As we are well aware of the fact that statemachines are related to its owning classes, a simple formalization can be derived. According to the UML metamodel, a statemachine is inherited from class [12] and thus can own a constraint. The context chosen therefore is statemachine.

Testing will be done via the VOD UML model (Figure 2.3), as it fulfills the requirements by owning at least one statemachine. The complete implementation again can be looked up in Appendix A.7.

Formalization

Statemachine must be assigned to its corresponding class.

Transformation

Nothing special except a simple owner-based validation happens during the transformation. The owner of the statemachine already being a class makes it relatively easy to perform this consistency check. For the sake of completeness, the transformation console output is shown in 4.34.

```
ADD ownedRule: 'Statemachine must be assigned to its corresponding class
    .'
TEST: model transformation successful ...
```

Listing 4.34: Transformation Console Output: Statemachine - Class.

Validation

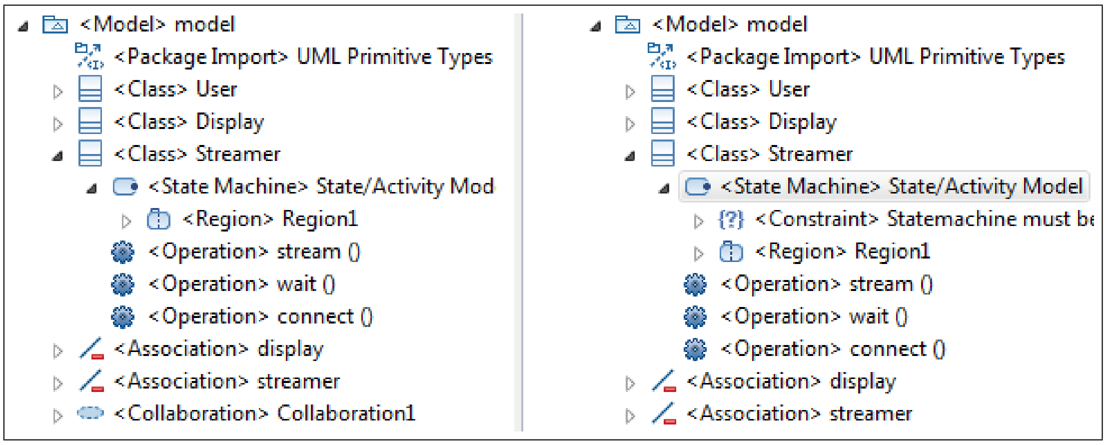
Like the transformation part, the validation can be expressed via a short OCL expression shown in Listing 4.35.

```
self.owner.oclIsTypeOf(Class)
```

Listing 4.35: OCL Expression: Statemachine - Class.

```
Evaluating:
self.owner.oclIsTypeOf(Class)
Results:
true
```

Listing 4.36: OCL Validation: Statemachine - Class.



```
<Model> model
    <Package Import> UML Primitive Types
    <Class> User
    <Class> Display
    <Class> Streamer
        <State Machine> State/Activity Mod
            <Region> Region1
        <Operation> stream ()
        <Operation> wait ()
        <Operation> connect ()
    <Association> display
    <Association> streamer
    <Collaboration> Collaboration1
```

```
<Model> model
    <Package Import> UML Primitive Types
    <Class> User
    <Class> Display
    <Class> Streamer
        <State Machine> State/Activity Model
            <Constraint> Statemachine must b
            <Region> Region1
        <Operation> stream ()
        <Operation> wait ()
        <Operation> connect ()
    <Association> display
    <Association> streamer
```

Figure 4.7: VOD UML Model Editor Validation.

4.1.7 Statemachine - Pseudostate

Another characteristic of statemachines is the existence of at least one initial pseudostate within a region. Truly, the UML metamodel only limits the amount of pseudostates to a maximum of one [12]. Indeed one could simply argue that the instance of any class type might be already in one of the possible states when instantiated. For this thesis, a experimental approach sometimes led to more specific and constrained design rules, which do not always conform to the UML metamodel. Nevertheless, as the region is one of multiple potential statemachine representations of a class, region is chosen as context.

The VOD UML model (Figure 2.3) provides the necessary existence of at least one region and therefore was used for validation. The exact module implementation is available in Appendix A.8.

Formalization

Statechart diagram must have an initial pseudostate.

Transformation

Corresponding to the preceding scenarios, constraints are added in an usual fashion and possibly missing pseudostates are added to the current region processed by the transformation (see Listing 4.37).

```
ADD ownedRule: ’Statechart region diagram must have an initial
    pseudostate.’
TEST: model transformation successful ...
```

Listing 4.37: Transformation Console Output: Statemachine - Pseudostate.

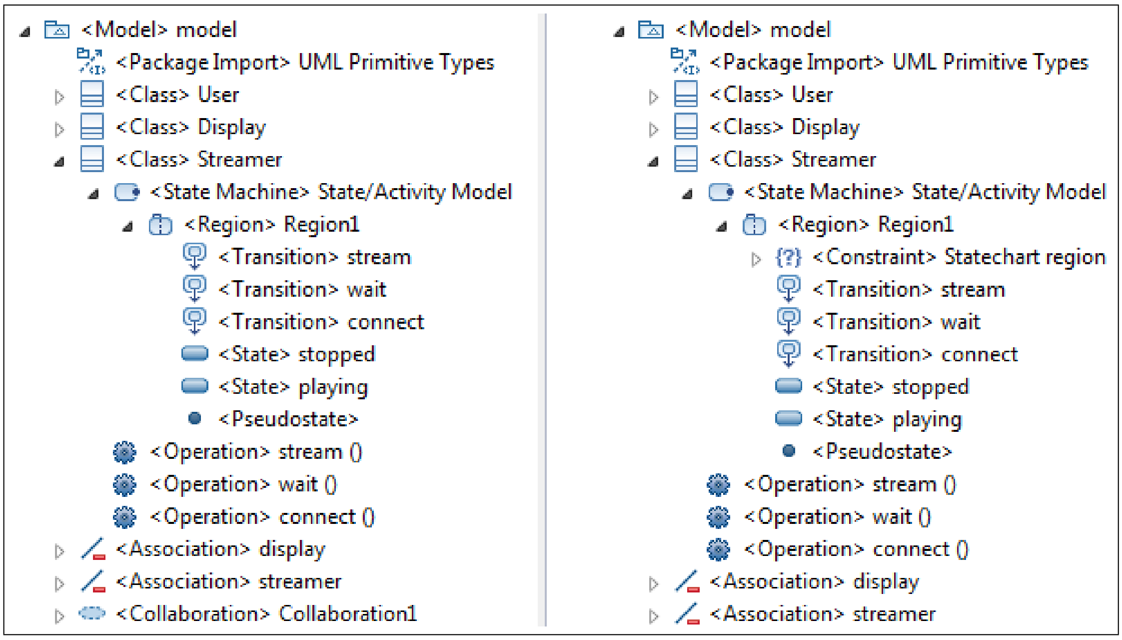
Validation

The rewritten this design-rule as OCL expression is shown in Listing 4.38.

```
self.ownedMember->select(oclIsTypeOf(Region)).ownedMember->exists(
    oclIsTypeOf(Pseudostate))
```

Listing 4.38: OCL Expression: Statemachine - Pseudostate.

In Figure 4.8 and Listing 4.39, no new pseudostate was added. Corresponding to the OCL expression, if a pseudostate already exist, none is added. But in the case of multiple pseudostates owned by one region, no violation is triggered too.



- <Model> model
    - <Package Import> UML Primitive Types
    - <Class> User
    - <Class> Display
    - <Class> Streamer
        - <State Machine> State/Activity Model
            - <Region> Region1
                - <Transition> stream
                - <Transition> wait
                - <Transition> connect
                - <State> stopped
                - <State> playing
                - <Pseudostate>
        - <Operation> stream ()
        - <Operation> wait ()
        - <Operation> connect ()
    - <Association> display
    - <Association> streamer
    - <Collaboration> Collaboration1

- <Model> model
    - <Package Import> UML Primitive Types
    - <Class> User
    - <Class> Display
    - <Class> Streamer
        - <State Machine> State/Activity Model
            - <Region> Region1
                - <Constraint> Statechart region
                - <Transition> stream
                - <Transition> wait
                - <Transition> connect
                - <State> stopped
                - <State> playing
                - <Pseudostate>
        - <Operation> stream ()
        - <Operation> wait ()
        - <Operation> connect ()
    - <Association> display
    - <Association> streamer

Figure 4.8: VOD UML Model Editor Validation.

```
Evaluating:
self.ownedMember->select(oclIsTypeOf(Region)).ownedMember->exists(
    oclIsTypeOf(Pseudostate))
Results:
true
```

Listing 4.39: OCL Validation: Statemachine - Class.

4.1.8 Association - Message

The eighth scenario describes the opposite of Subsection 4.1.5. Whereas for each message an association has to exist, the other way around can be stated as well. Despite describing sequence diagrams as a subset of the class diagram, one could agree upon having an association between two classes, thus at least one message between the two corresponding lifelines has to exist. For the sake of completeness, this would make sense when we are going to model the whole system as sequence diagram. According to the UML metamodel, it has to be mentioned that it is in fact not necessary. However, in rare cases, this rule could be useful so we devote one of the nine rules as its counterpart to another one. In terms of finding an appropriate context, we know by fact that there is at maximum one constraint added to each association. Therefore and without a doubt, the association fits the context for the matched rule, the constraint generated and the final OCL expression.

As the VOD AR UML model was chosen for the counterpart scenario, (Figure 2.4 and Figure 2.5) is selected again. The full source code is appended in Appendix A.9.

Formalization

For each association, the corresponding message must exist.

Transformation

No special helpers were used, but for the generation of instances, message and MOS called rules were defined. This is necessary as a message depends on its MOSs which attaches it to the sender as well as receiver lifelines. In the case the association derived message is missing, a new message and two new MOSs have to be generated. Since the context is of type association and only one constraint will be added to it, constraints can be appended directly in the *to* section during the transformation shown in Listing 4.40.

```
to
   t: UML2!Association (
     -- keep association properties

     -- add constraint
     ownedRule <- s.ownedRule -> append(thisModule.NewOwnedRule(s,
         c08Name, c08Expr, 'OCL'))
   )
```

Listing 4.40: ATL Rule: to Section.

The remaining *do* section of the matched rule checks whether each association member end does exist. Further, sender and receiver events of lifelines have to do as well. In order to satisfy this constraint, such message - connecting the class lifelines - must exist or be generated.

For this particular UML model under test, the output console in Listing 4.41 shows two new generated messages. To understand this, we have to look at the sequence diagram. In the sequence diagram, three lifelines do exist: `Display`, `Service` and `Streamer`. According to the associations, two messages between `Service` and `Display` as well as

`Streamer` and `Display` are generated (indeed with distinctive identifiers).

```
ADD ownedRule: 'For the association display, the corresponding message
    must exist.'
ADD ownedRule: 'For the association streamer, the corresponding message
    must exist.'
ADD ownedRule: 'For the association server, the corresponding message
    must exist.'
ADD ownedRule: 'For the association display, the corresponding message
    must exist.'
ADD message: 'display'
ADD message occurrence specification: 'display_Send'
ADD message occurrence specification: 'display_Receive'
ADD message: 'display'
ADD message occurrence specification: 'display_Send'
ADD message occurrence specification: 'display_Receive'
TEST: model transformation successful ...
```

Listing 4.41: Transformation Console Output: Association - Message.

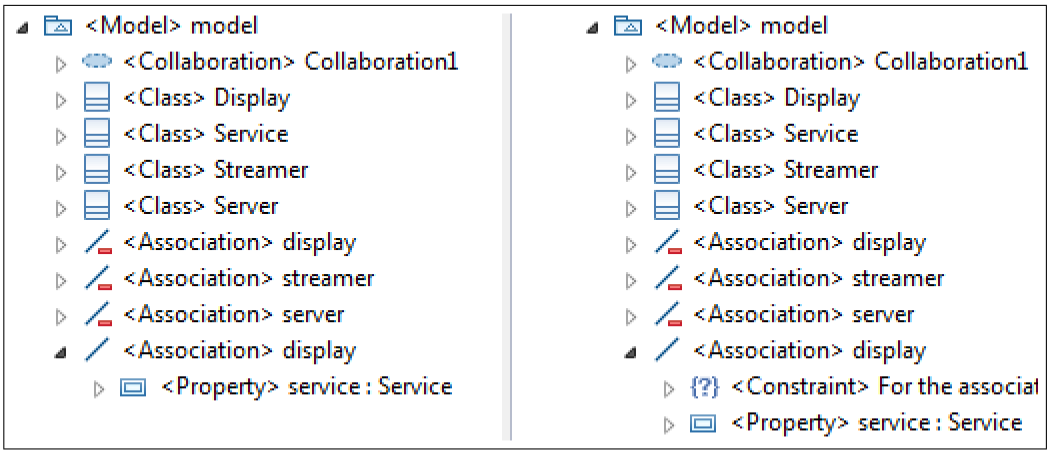
Validation

Investigating the generated OCL expression in Listing 4.42, `snd` stands for sender lifeline and `rcv` for receiver lifeline. In the nested `let` statements both are evaluated first. The last line then verifies if a message connected to both exists.

```
let snd: Lifeline = Lifeline.allInstances()->select(represents.type =
    self.memberEnd->at(1).type) in
let rcv: Lifeline =Lifeline.allInstances()->select(l | l.represents.type
    = self.memberEnd->at(2).type) in
not Message.allInstances()->exists(receiveEvent = rcv)
```

Listing 4.42: OCL Expression: Association - Message.

Figure 4.9 shows both messages as well as MOFs being added to the model on the right.



<Model> model
  <Collaboration> Collaboration1
  <Class> Display
  <Class> Service
  <Class> Streamer
  <Class> Server
  <Association> display
  <Association> streamer
  <Association> server
  <Association> display
    <Property> service : Service

<Model> model
  <Collaboration> Collaboration1
  <Class> Display
  <Class> Service
  <Class> Streamer
  <Class> Server
  <Association> display
  <Association> streamer
  <Association> server
  <Association> display
    <Constraint> For the associat
    <Property> service : Service

Figure 4.9: VOD AR UML Model Editor Validation.

The actual validation for the expressed OCL statement below in Listing 4.43 evaluates to `true`.

```
Evaluating:
let snd: Lifeline = Lifeline.allInstances()->select(represents.type =
    self.memberEnd->at(1).type) in
let rcv: Lifeline =Lifeline.allInstances()->select(l | l.represents.type
    = self.memberEnd->at(2).type) in
not Message.allInstances()->exists(receiveEvent = rcv)
Results:
true
```

Listing 4.43: OCL Validation: Association - Message.

4.1.9 Activity - Operation

With similar fashion to messages, activities in statemachines are considered operations in the corresponding class diagram. Thus the context is of type class, alike the scenario for messages and operations. Activities are optional behaviors and are distinguished by its preceding tag which either is `/entry`, `/exit` or `/do`.

The VOD AR UML model (Figure 2.4 and Figure 2.5) contains each of those three activities once. The exact implementation can be looked up in Appendix A.10.

Formalization

Activity must be represented by an operation.

Transformation

Alongside the message - operation scenario, and for the class in context, each activity must be represented by its corresponding operation. The console output shows the added operations as well as the appended constraint for the `Streamer` class containing both statemachines.

```
ADD operation: 'someActivity'
ADD operation: 'otherActivity'
ADD operation: 'anotherActivity'
ADD ownedRule: 'Activity must be represented by an operation.'
TEST: model transformation successful ...
```

Listing 4.44: Transformation Console Output: Activity - Operation.

Validation

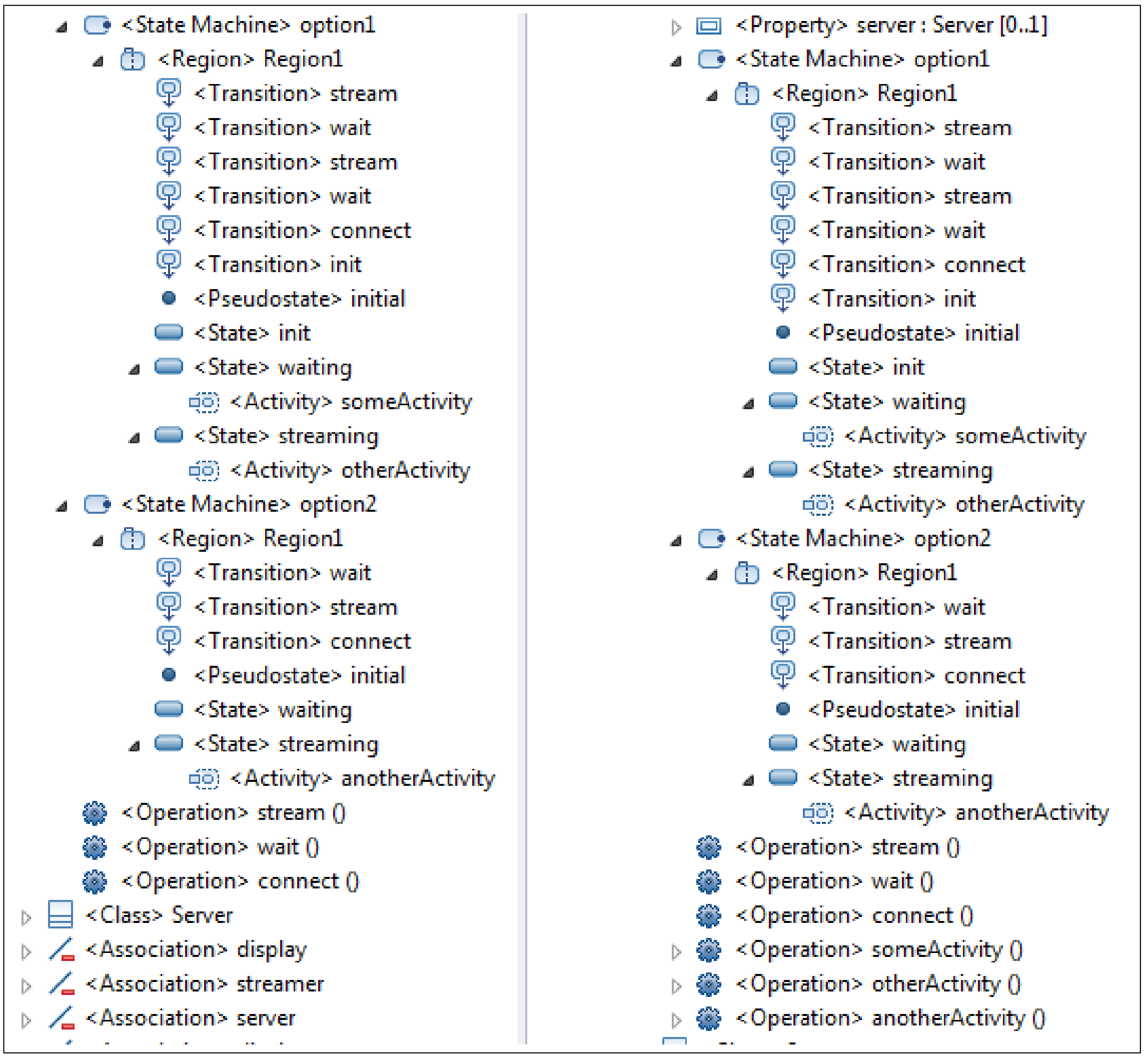
For each class (and indeed for possible superclasses), the constraint rule were built via concatenation of all occurring activity names (see Listing 4.45).

```
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='someActivity') and
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='otherActivity') and
```

```
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='anotherActivity')
```

Listing 4.45: OCL Expression: Activity - Operation.

Figure 4.10 shows the UML model before (left) and after (right) the transformation.



- <State Machine> option1
  - <Region> Region1
    - <Transition> stream
    - <Transition> wait
    - <Transition> stream
    - <Transition> wait
    - <Transition> connect
    - <Transition> init
    - <Pseudostate> initial
    - <State> init
    - <State> waiting
      - <Activity> someActivity
    - <State> streaming
      - <Activity> otherActivity
- <State Machine> option2
  - <Region> Region1
    - <Transition> wait
    - <Transition> stream
    - <Transition> connect
    - <Pseudostate> initial
    - <State> waiting
    - <State> streaming
      - <Activity> anotherActivity
  - <Operation> stream ()
  - <Operation> wait ()
  - <Operation> connect ()
- <Class> Server
- <Association> display
- <Association> streamer
- <Association> server

- <Property> server : Server [0..1]
- <State Machine> option1
  - <Region> Region1
    - <Transition> stream
    - <Transition> wait
    - <Transition> stream
    - <Transition> wait
    - <Transition> connect
    - <Transition> init
    - <Pseudostate> initial
    - <State> init
    - <State> waiting
      - <Activity> someActivity
    - <State> streaming
      - <Activity> otherActivity
- <State Machine> option2
  - <Region> Region1
    - <Transition> wait
    - <Transition> stream
    - <Transition> connect
    - <Pseudostate> initial
    - <State> waiting
    - <State> streaming
      - <Activity> anotherActivity
- <Operation> stream ()
- <Operation> wait ()
- <Operation> connect ()
- <Operation> someActivity ()
- <Operation> otherActivity ()
- <Operation> anotherActivity ()

Figure 4.10: VOD AR UML Model Editor Validation.

As expected, Listing 4.46 evaluates to `true` after adding the operations to the class diagram.

```
Evaluating:
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='someActivity') and
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='otherActivity') and
self.inheritedMember->select(oclIsTypeOf(Operation))->union(self.
    ownedOperation)->exists(name='anotherActivity')
Results:
true
```

Listing 4.46: OCL Validation: Activity - Operation.

4.2 Usage Documentation

This section covers a detailed view on prerequisites, setup, usage and validation for the thesis practical work.

4.2.1 Prerequisites

Eclipse

Eclipse Modeling Tools 1.5.1 (Juno Service Release 1) available for download at `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr1`. This package contains the most important plug-ins for MDSD in Eclipse framework.

ATL

ATL SDK - ATLAS Transformation Language SDK 3.3.1 available through the Eclipse menu: *Help* → *Install Modeling Components* or for download at `http://wiki.eclipse.org/ATL/User_Guide_-_Installation#Install_ATL`. ATL SDK is the prerequisite Eclipse plug-in on which EMVTVM is built upon.

ATL EMFTVM

EMF Transformation Virtual Machine 3.4.0 available through its update site: `http://soft.vub.ac.be/eclipse/update-3.7/`. The extended VM for EMF model transformation developed at the Department of Computer Science of the Vrije Universiteit Brussel (VUB).

Papyrus

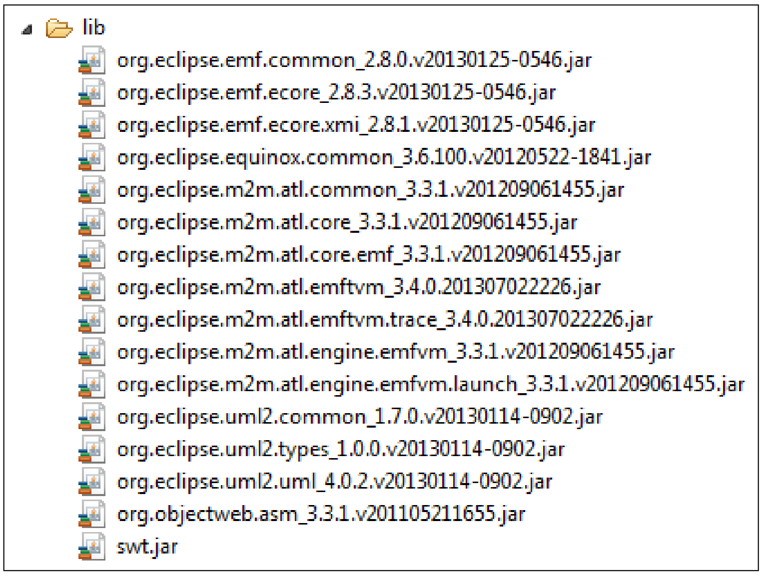
Papyrus SDK Binaries (Incubation) 0.9.2 available through the Eclipse menu: *Help* → *Install Modeling Components* or for download at `http://www.eclipse.org/papyrus/downloads/`.

Xtext OCL Console for UML Model Editor

OCL Examples and Editors 3.2.2 available through the Eclipse menu: *Help* → *Install Modeling Components* or for download at `http://www.eclipse.org/modeling/mdt/downloads/?project=ocl`.

Libraries for Standalone Execution

In addition to Eclipse framework as well as the additional plug-ins, the Eclipse libraries shown in Figure 4.11 are required for ATL EMFTVM standalone execution [3]. For convenient transformation execution, a simple SWT GUI was added and therefore the SWT library is needed as well.



- lib
  - org.eclipse.emf.common_2.8.0.v20130125-0546.jar
  - org.eclipse.emf.ecore_2.8.3.v20130125-0546.jar
  - org.eclipse.emf.ecore.xmi_2.8.1.v20130125-0546.jar
  - org.eclipse.equinox.common_3.6.100.v20120522-1841.jar
  - org.eclipse.m2m.atl.common_3.3.1.v201209061455.jar
  - org.eclipse.m2m.atl.core_3.3.1.v201209061455.jar
  - org.eclipse.m2m.atl.core.emf_3.3.1.v201209061455.jar
  - org.eclipse.m2m.atl.emftvm_3.4.0.201307022226.jar
  - org.eclipse.m2m.atl.emftvm.trace_3.4.0.201307022226.jar
  - org.eclipse.m2m.atl.engine.emfvm_3.3.1.v201209061455.jar
  - org.eclipse.m2m.atl.engine.emfvm.launch_3.3.1.v201209061455.jar
  - org.eclipse.uml2.common_1.7.0.v20130114-0902.jar
  - org.eclipse.uml2.types_1.0.0.v20130114-0902.jar
  - org.eclipse.uml2.uml_4.0.2.v20130114-0902.jar
  - org.objectweb.asm_3.3.1.v201105211655.jar
  - swt.jar

Figure 4.11: Libraries.

4.2.2 Project Setup

The complete framework is built from the Papyrus project (for the UML model representation and creation), the ATL project (for transformation) and the Java project (for the programmatical launch via SWT GUI). Figure 4.12 displays the three projects in a hierarchical tree view.

Papyrus Project

The *Models* project includes the UML metamodel and the Papyrus input models expanded in path *Models/papyrus/models/...*.

ATL Project

The *Transformations* project comprises all ATL transformations: the *inplace* folder representing the used transformation modules using refinement mode of the EMFTVM. Since ATL modules are compiled through their VM to byte-code, the ATL nature must be set for the project. For the EMFTVM, actually a JIT-Compiler handles the complex compilation process. This results in a smaller binary file (*.emftvm) in contrast to the larger ASM files (*.asm) of the original ATL transformations [3].

Java Project

For simple transformations, the ATL wizard (provided by the Eclipse plug-in itself) might fulfill all needs. In order to maintain incremental transformation support, or package the transformation in an existing framework, programmatical launch with Java has to be considered. Therefore, *EMFTVMLauncher* provides a constructor including all possible execution parameters. While one can execute this class as *Java Application*, a more convenient way of execution is provided via the *Standard Widget Toolkit (SWT)* [2] derived *Graphical User Interface (GUI)* [3] implementation by the *Window* class.

As mentioned above and in Subsection 4.2.1, additional libraries are necessary for standalone use. In this framework the user library *uml2uml* was created.

4.2.3 Execution

To perform the actual transformation, class *Window* must be executed as *Java Application*. The GUI shown in Figure 4.13 allows the specification of the ATL module file, the input UML model file and an optional output UML model file. In order to prohibit overwriting the input model after transformation, an optional output model file path can be specified. In the current state, the file paths are processed as Microsoft Windows delimiters. The console output within Eclipse framework gives immediate information concerning new UML elements being added during the transformation.

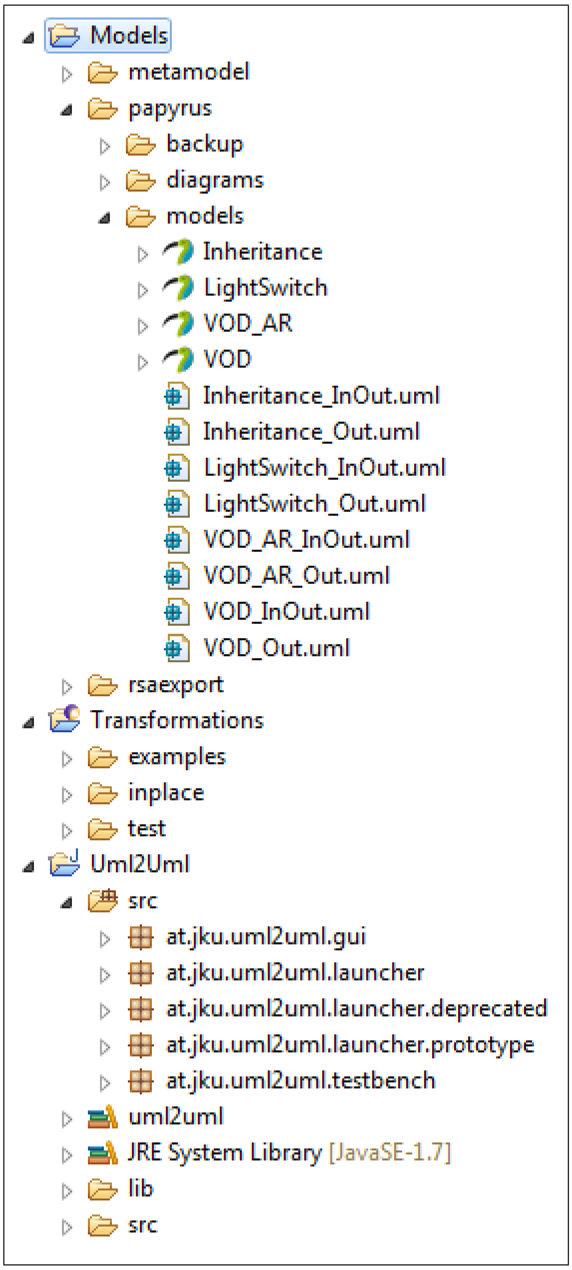
4.2.4 Validation

The manual validation is executed with the support of Xtext OCL console in UML Model Editor inside Eclipse framework. The Xtext OCL interpreter can be opened via right clicking and choosing the *Show Xtext OCL console* command. To specify the context of the OCL expression, an UML element must be selected. A given OCL expression, e.g. the one generated through transformation, can then be processed within the console.

---

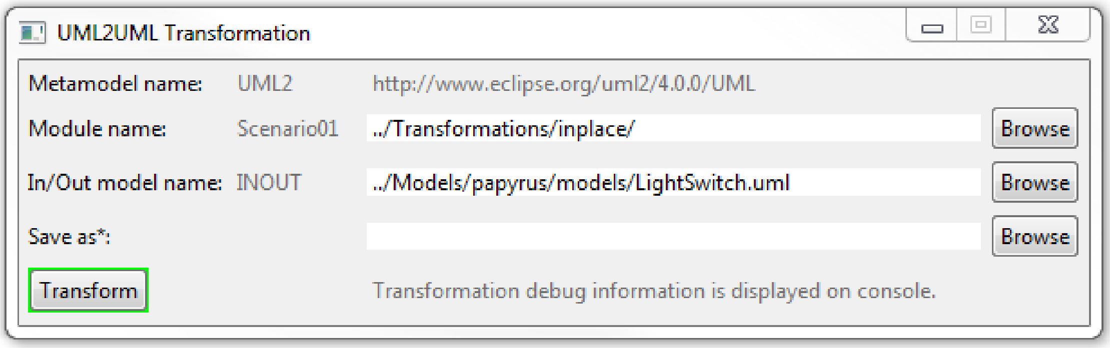
[2] Online at: `http://www.eclipse.org/swt/`.

[3] Online at: `http://en.wikipedia.org/wiki/Graphical_user_interface`.



- Models
  - metamodel
  - papyrus
    - backup
    - diagrams
    - models
      - Inheritance
      - LightSwitch
      - VOD_AR
      - VOD
      - Inheritance_InOut.uml
      - Inheritance_Out.uml
      - LightSwitch_InOut.uml
      - LightSwitch_Out.uml
      - VOD_AR_InOut.uml
      - VOD_AR_Out.uml
      - VOD_InOut.uml
      - VOD_Out.uml
  - rsaexport
- Transformations
  - examples
  - inplace
  - test
- Uml2Uml
  - src
    - at.jku.uml2uml.gui
    - at.jku.uml2uml.launcher
    - at.jku.uml2uml.launcher.deprecated
    - at.jku.uml2uml.launcher.prototype
    - at.jku.uml2uml.testbench
  - uml2uml
  - JRE System Library [JavaSE-1.7]
  - lib
  - src

Figure 4.12: Eclipse Project Explorer.



UML2UML Transformation

Metamodel name: UML2 http://www.eclipse.org/uml2/4.0.0/UML

Module name: Scenario01 ../Transformations/inplace/

In/Out model name: INOUT ../Models/papyrus/models/LightSwitch.uml

Save as*:

Transformation debug information is displayed on console.

Figure 4.13: Transformation Execution SWT GUI.

Chapter 5

Related Work

Let us take a look at some similar work done in the last decade. Started as the *ATLAS Group* and nowadays known under the name of *AtlanMod* [1], the team contributed important work on scientific research concerning model-to-model transformation in MDE. The ATL and their continued work on evolving the language is by far their biggest achievement. Alongside with this thesis in [25], Jouault and Bzivin proposed a metamodel-independent OCL validation approach through the ATL. As the metamodel, they defined a class diagram-like model, which is in fact a subset of the UML metamodel. Besides extending the OCL for meaningful annotations, such as covering informal constraint descriptions, OCL expressions were verified. A set of verified expressions is called *Diagnostics*. For the purpose of representing the actual verification result as a model, Diagnostics conforms to its own metamodel. Moreover, they translated each OCL constraint into the corresponding ATL rule, just like we have seen for the 9 scenarios in Chapter 4 of this thesis. Actually, they only provided information about constraint violation, whereas this thesis presented partial fixing in addition to generating OCL expressions.

In *Identification and Check of Inconsistencies between UML Diagrams* [26], Liu targeted inconsistencies between different UML diagrams, but described in Prosa only. Although no formal validation or transformation is used, finding alone the semantic connections between multiple UML diagrams is not trivial at all. For our work, the preliminary work of Liu provided initial and useful insights concerning constraint scenario formalization, as well as thoughts on dealing with non-deterministic actions for fixing those inconsistencies.

---

[1] Online at: `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`.

Model transformations and constraint generations can cause a lot of challenges such as non-deterministic choices for fixing inconsistencies, incremental rule execution dependencies, possible model instantiations due to complex metamodels, bidirectional execution and many more. In [23], Demuth et al. illustrated a partial solution via constraint-driven modeling and ATL-like transformations. Specifying constraints to narrow down or control the validation space, and because of the fact that constraints do not interfere with each other concerning the order of execution, the problems mentioned above can be solved.

Providing guidance through immediate response to changes, made by the designer, can solve the fixing part of non-deterministic choices as well. In general, no algorithm would be able to make the right decisions when the problem is of ambiguous nature. The work we presented stops fixing inconsistencies as soon as multiple actions would be possible. In [27] Egyed presents the *UML/Analyzer* framework which deals with this aspects through profiling techniques. The user then is presented a set of actions to choose from, which as a result resolve the issue of multiple choices. The paper clearly states that the designer is responsible for picking the right choice in order to conduct the fix. Not only consistency contributes to a good model.

On the contrary, Egyed et al. discusses automated support for fixing inconsistencies in [28]. Not only the fix alone is considered satisfying, but also the impact of the fix is measured. In order to select the best fix, all possible fixes (e.g. a missing operation is added, an existing one renamed, or the counterpart of the operation just deleted) are executed and only a fix which does not cause any new inconsistencies counts as the best fix.

As a subsequent successor to [27], Egyed and Reder again developed *Model/Analyzer: A Tool for Detection, Visualizing and Fixing Design Errors in UML* [17]. Now based on the *Rational Software Modeler (RSM)* [2], Model/Analyzer excels at its customizablity such as design rule creation, in context validation and, most importantly, automated and incremental feedback.

---

[2] Now included in *Rational Software Architect (RSA)* and online available at: `http://www.ibm.com/developerworks/rational/products/rsa/`.

Chapter 6

Conclusions and Future Work

A transformation framework for automatic, partial and incremental fixing of inconsistencies was presented. In addition, generated OCL expressions validated the performed actions onto the UML model. For convenient usage, a GUI encapsulates the programmatically launched transformation.

Analogous to similar work, confronted in Chapter 5, the difficulty of automatically fixing inconsistencies with completeness, still remains. Overcoming the complexity of non-deterministic fixing choices, is one of the biggest problems in this domain which still has to be solved. Nevertheless, showing the success of the set of scenarios implemented and discussed, this work is considered beneficial in the fields of MDE, which is very alive within the Eclipse community. Moreover, the interoperability needed for Open-source approaches is achieved by the anticipated Eclipse plug-in development teams.

For future work, the addition of building a subset of fixing choices and would be beneficial. Letting the user decide based on outcome information would constitute significant improvement. For this in particular, one would have to extend UML model interactivity through the *Adapter/Observer* design pattern. Triggered by change, transformation rules can be built and assembled alike [29] to fulfill the change's implications. Although the ATL currently supports ATL module import as well as rule inheritance [3], dependencies during rule execution may complicate things. We are not aware of any UML modelling frameworks, where the graphical representation is built up based on the UML file alone. Hence, bidirectionality is hard to achieve, although refining mode in-place transformation already being supported in ATL EMFTVM [3]. But the performance boost of not re-transforming (copying) unmatched model elements, does significantly secure scalability for even large UML models.

Other than the major potential improvements pointed out, minor enhancements such as simplifying some OCL expressions and rearranging the UML element in context could be done to ease validation. On the one hand, constraints have their element names looked up during transformation, but on the other hand they are implemented only as context-dependent expressions. The OCL validation was conducted manually to show the work's correctness, but still, to cover all scenarios for the UML models under test consumes time. One might consider automatic validation, although the transformation rules actually do represent the expression - only rewritten into a similar but yet another language.

Appendix A

Source Code

A.1 Sequence to Class Diagram

```
-- (c) Stefan Luger 2013
-- Transforms UML2 Sequence diagram to UML2 Class diagram
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Seq2Class;
create OUT: UML2 from IN: UML2;

helper def: getLifelines(): Sequence(UML2!"uml::Lifeline") =
  UML2!"uml::Lifeline".allInstances();

helper def: getConstraints(): Sequence(UML2!"uml::Constraint") =
  UML2!"uml::Constraint".allInstances();

helper def: getMessages(): Sequence(UML2!"uml::Message") =
  UML2!"uml::Message".allInstances();

-- for each message create tuple sets of lifelineSend and
    lifelineReceived
helper def: getAssociations(): Sequence(OclAny) =
  let rcv: OclAny =
    thisModule.getReceiveLifelines()
  in
    let snd: OclAny =
      thisModule.getSendLifelines()
    in
      rcv -> iterate(i; assSeq: Sequence(UML2!"uml::Lifeline") =
          Sequence {} |
          assSeq.append(Sequence{i,
              snd -> at(assSeq.size() + 1)}));

helper def: getReceiveLifelines(): Sequence(UML2!"uml::Lifeline") =
  thisModule.getMessages() -> collect(re | re.receiveEvent.covered).
      first();

helper def: getSendLifelines(): Sequence(UML2!"uml::Lifeline") =
  thisModule.getMessages() -> collect(se | se.sendEvent.covered).first()
      ;

rule Model {
  from
```

```
    s: UML2!"uml::Model"
  to
    t: UML2!"uml::Model" (
      name <- s.name,
      ownedRule <- s.ownedRule,
      packagedElement <- thisModule.getLifelines() -> union(thisModule.
          getConstraints()) -> union(thisModule.getAssociations() ->
          iterate(iter; a: Sequence(UML2!"uml::Association") = Sequence
              {} | a.
          append(thisModule.Association(iter.at(1), iter.at(2)))))
    )
}

unique lazy rule Association {
  from rcv: UML2!"uml::Lifeline", snd: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Association" (
      name <- rcv.name + '_' + snd.name,
      --   memberEnd <-
      ownedEnd <- Sequence{thisModule.AssociationOwnedEnd(rcv, snd)}
    )
  do {
    t; -- return generated association
  }
}

lazy rule AssociationOwnedEnd {
  from rcv: UML2!"uml::Lifeline", snd: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Property" (
      name <- snd.name,
      type <- snd,
      lowerValue <- thisModule.LiteralInteger(1),
      upperValue <- thisModule.LiteralUnlimitedNatural(1)
    )
  do {
    t;
  }
}

lazy rule ClassOwnedAttributeAssociation {
  from rcv: UML2!"uml::Lifeline", snd: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Property" (
      name <- snd.name,
      type <- snd,
      association <- thisModule.Association(rcv, snd),
      lowerValue <- thisModule.LiteralInteger(1),
      upperValue <- thisModule.LiteralUnlimitedNatural(1)
    )
  do {
    t;
  }
}

rule LiteralInteger (v: Integer) {
  to
    t: UML2!"uml::LiteralInteger" (
      value <- v
    )
 do {
    t;
  }
}

rule LiteralUnlimitedNatural (v: Integer) {
  to
    t: UML2!"uml::LiteralUnlimitedNatural" (
      value <- v
    )
 do {
    t;
  }
}
```

```

rule OpaqueExpression {
  from
    s: UML2!"uml::OpaqueExpression"
  to
    t: UML2!"uml::OpaqueExpression" (
      name <- s.name,
      visibility <- s.visibility,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
      body <- s.body,
      language <- s.language,
      behavior <- s.behavior
    )
}

rule Message2Operation {
  from
    s: UML2!"uml::Message"
  to
    t: UML2!"uml::Operation" (
      name <- s.name
    )
}

rule Lifeline2Class {
  from
    s: UML2!"uml::Lifeline"
  to
    t: UML2!"uml::Class" (
      name <- s.name,
      visibility <- s.visibility,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
      ownedOperation <- thisModule.getMessages(),
      ownedAttribute <- let assList: Sequence(OclAny) =
          thisModule.getAssociations()
        in
          if assList -> isEmpty() then
            Sequence {}
          else
            let a: Sequence(OclAny) =
              assList -> select(a | if a -> at(1) = s then
                    true
                  else
                    true
                  endif)
            in
              if a -> isEmpty() then
                Sequence {}
              else
                Sequence {}.append(thisModule.
                    ClassOwnedAttributeAssociation(a -> flatten() ->
                    at(1), a -> flatten() -> at(2)))
              endif
          endif
    )
}

rule Constraint {
  from
    s: UML2!"uml::Constraint"
  to
    t: UML2!"uml::Constraint" (
      name <- s.name,
      visibility <- s.visibility,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
```

```
      constrainedElement <- s.constrainedElement,
      specification <- s.specification
    )
}
```

Listing A.1: Seq2Class.atl

A.2 Constraint-driven Scenarios

```
-- (c) Stefan Luger 2013
-- Each message must be represented by an operation and inside the
   corresponding class
    -- hierarchy.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario01;
create OUT: UML2 refining IN: UML2;

-- helpers

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getReceiverLifelineClass(m: UML2!Message): UML2!Class =
  UML2!Lifeline.allInstancesFrom('INOUT') -> select(l | l.coveredBy ->
      select(i | i.
      oclIsTypeOf(UML2!MessageOccurrenceSpecification)) -> exists(e | e
         = m.
      receiveEvent)) -> first().represents.type;

helper def: getMessagesByClass(cl: UML2!Class): Sequence(UML2!Message) =
  UML2!Message.allInstancesFrom('INOUT') -> select(m | thisModule.
      getReceiverLifelineClass(m) = cl);

-- new operation constructor alternative
rule NewOperation (oStr: String, cStr: String, owner: OclAny){
  using {
    o: UML2!Operation = UML2!Operation.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    o.name <- oStr -> debug('ADD operation');
    if (owner <> OclUndefined) o.class <- owner;
    o; -- return operation
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
    String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
        INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
```

```
}

-- for each message, look up missing operations in inheritance hierarchy
rule Class {
  from
    s: UML2!Class (
      s.oclIsTypeOf(UML2!Class)
    )
  using {
    c01Name: String = 'For the class \'' + s.name + '\', each message
        must be' + '' +
         ' represented by an operation and inside the corresponding class
             ' + '' +
         ' hierarchy.';
    c01Expr: String = OclUndefined;
    c01Elements: Sequence(UML2!Message) = OclUndefined;
    newOps: Sequence(UML2!Message) = thisModule.getMessagesByClass(s) ->
         debug('ConcurrentModificationException Fix') -> select(m | not s
             .
         ownedOperation -> exists(o | o.name = m.name));
  }
  to
    t: UML2!Class (
      -- keep class properties
    )
  do {
      -- add missing operations
    for (m in newOps) {
         -- when there is no super class, add operation to class
         if (not s.allOwnedElements() -> exists(g | g.
              oclIsTypeOf(UML2!Generalization))) {
           thisModule.NewOperation(m.name, '', s);
         }
         -- otherwise add operation to model, in case it doesn't exist
             yet
         else if (UML2!Operation -> allInstancesFrom('INOUT') -> select(o
              | o.
              owner = OclUndefined and o.ownedComment -> exists(oc | oc.
                 body =
              c01Name)) -> isEmpty()) {
           thisModule.NewOperation(m.name, c01Name, OclUndefined);
         }
      } -- get all messages for constraint expression
      c01Elements <- thisModule.getMessagesByClass(s);

    -- for each operation, build constraint
    if (c01Elements -> size() > 0) {
      c01Expr <- 'self.inheritedMember->select(oclIsTypeOf(Operation))->
          union(self.'
           + 'ownedOperation)->exists(name=\'' + c01Elements.first().
               name +
           '\')';

      c01Elements <- c01Elements -> subSequence(2, c01Elements -> size()
          );
      for (o in c01Elements) {
         c01Expr <- c01Expr.concat(' and self.' +
              'inheritedMember->select(oclIsTypeOf(Operation))->union(
                 self.'
              + 'ownedOperation)->exists(name=\'' + o.name + '\')');
      } -- add constraint to class
        if (not s.allOwnedElements() -> select(c | c.
           oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c01Name)
                and
           s.oclIsTypeOf(UML2!Class)) {
         thisModule.NewOwnedRule(s, c01Name, c01Expr, 'OCL');
      }
    }
  }
}
```

Listing A.2: Scenario01.atl

```
-- (c) Stefan Luger 2013
-- For each lifeline, a corresponding class must exist.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario02;
create OUT: UML2 refining IN: UML2;

-- helpers

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getLifelineClass(l: UML2!Lifeline): UML2!Class =
  if (l.represents = OclUndefined) then
    OclUndefined
  else
    if (l.represents.type = OclUndefined) then
      OclUndefined
    else
      l.represents.type
    endif
  endif;

-- new lifeline class link property constructor
rule NewLifelineClassLinkProperty (name: String, cl: UML2!Class, o: UML2!
    Collaboration) {
  using {
    p: UML2!Property = UML2!Property.newInstanceIn('INOUT') -> debug('
        ADD' + '' +
         ' property');
  }
  do{
    p.name <- name;
    p.type <- cl;
    o.ownedAttribute <- o.ownedAttribute -> append(p);
    p; -- return property
  }
}

-- new class constructor alternative
rule NewClass (name: String, abst: Boolean) {
  using {
    cl: UML2!Class = UML2!Class.newInstanceIn('INOUT');
  }
  do{
    cl.name <- name;
    cl.isAbstract <- abst;
    thisModule.getModel().packagedElement <- thisModule.getModel().
        packagedElement ->
         append(cl);
    cl; -- return class
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
     String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
        INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
```

```
      c; -- return constraint
    }
}

-- matched rules
rule Lifeline {
  from
    s: UML2!Lifeline (
      s.oclIsTypeOf(UML2!Lifeline)
    )
  using {
    c02Name: String = 'For each lifeline, a corresponding class must exist.';
    validLlName: String = s.name.at(1) -> toUpper() + s.name.substring(2);
    c02Expr: String = 'Lifeline.allInstances()->select(name = \'' + validLlName +
        '\').represents.type->notEmpty()';
    c02Owner: UML2!Class = OclUndefined;
    collab: UML2!Collaboration = s.owner.owner;
  }
  to
    t: UML2!Lifeline (
      -- lifeline must start with a capital character, in case of
      -- violation, change it
      name <- validLlName
    ) -- keep lifeline properties

  do {
    -- add class to model
    if (thisModule.getLifelineClass(s) = OclUndefined) {
      -- when class with the same name as the Lifeline does exist, but just isn't
          -- linked
      -- yet, set constraint owner
      -- otherwise, create new class
      if (thisModule.getModel().allOwnedElements() -> exists(cl | cl.
          oclIsTypeOf(UML2!Class) and cl.name = s.name)) {
        c02Owner <- thisModule.getModel().allOwnedElements() -> select(cl | cl.
            oclIsTypeOf(UML2!Class) and cl.name = s.name) -> debug('
            FOUND class');
      } else {
        -- no abstract class creation
        c02Owner <- thisModule.NewClass(s.name, false) -> debug('ADD class');
      }

      -- when there is no property for the represents attribute, add a new property
      if (s.represents = OclUndefined) {
        s.represents <- thisModule.NewLifelineClassLinkProperty(s.name.toLower(),
            c02Owner, collab);
      } else if (s.represents.type = OclUndefined) {
        s.represents.type <- c02Owner;
      }
    } else {
      c02Owner <- thisModule.getLifelineClass(s);
    }

    -- add constraint to lifeline
    if (not c02Owner -> allOwnedElements() -> select(c | c.
    oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c02Name) and c02Owner.
    oclIsTypeOf(UML2!Class)) {
      thisModule.NewOwnedRule(c02Owner, c02Name, c02Expr, 'OCL');
    }
  }
}
```

Listing A.3: Scenario02.atl

```
-- (c) Stefan Luger 2013
-- For each transition, a corresponding operation must exist.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario03;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getTransitionsByClass(cl: UML2!Class): Sequence(UML2!Transitions) =
  UML2!Transition.allInstancesFrom('INOUT') -> select(t | t.owner.owner.owner = cl);

-- new operation constructor alternative
rule NewOperation (oStr: String, cStr: String, owner: OclAny){
  using {
    o: UML2!Operation = UML2!Operation.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstance();
  }
  do{
    c.body <- cStr;
    o.name <- oStr -> debug('ADD operation');
    o.ownedComment <- Sequence{}.append(c);
    if (owner <> OclUndefined) o.class <- owner;
    o; -- return operation
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l: String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}

-- for each transition, look up missing operations in inheritance hierarchy
rule Class {
  from
    s: UML2!Class
  using {
    c03Name: String = 'For each transition, a corresponding operation must exist.';
    c03Expr: String = OclUndefined;
    c03Elements: Sequence(UML2!Operation) = OclUndefined;
    newOps: Sequence(UML2!Transition) = thisModule.getTransitionsByClass(s) ->
         select(tr | not s.ownedOperation -> exists(o | o.name = tr.name));
  }
  to
    t: UML2!Class (
      -- keep class properties
    )
  do {
```

```

    if (not thisModule.getTransitionsByClass(s) -> select(t | not s.
       ownedOperation ->
        exists(o | o.name = t.name)) -> isEmpty()) {
         -- add missing operations
      for (tr in newOps) {
        -- when there is no super class, add operation to class
        if (not s.allOwnedElements() -> exists(g | g.
            oclIsTypeOf(UML2!Generalization))) {
          thisModule.NewOperation(tr.name, ’’, s);
        }
        -- otherwise add operation to model, in case it doesn’t exist
           yet
        else if (UML2!Operation -> allInstancesFrom(’INOUT’) -> select(o
            | o.
            owner = OclUndefined and o.ownedComment -> exists(oc | oc.
               body =
            c03Name)) -> isEmpty()) {
          thisModule.NewOperation(tr.name, c03Name, OclUndefined);
        }
      }
    }
       -- get all operations for constraint expression
    c03Elements <- thisModule.getTransitionsByClass(s);--s.
       ownedOperation ->
        -- union(newOps);
    -- for each operation, build constraint
    if (c03Elements -> size() > 0) {
      c03Expr <- ’self.inheritedMember->select(oclIsTypeOf(Operation))->
         union(self.’
          + ’ownedOperation)->exists(name=\’’ + c03Elements.first().
             name
          + ’\’)’;

      c03Elements <- c03Elements -> subSequence(2, c03Elements -> size()
         );
      for (o in c03Elements) {
        c03Expr <- c03Expr.concat(’ and self.’ +
            ’inheritedMember->select(oclIsTypeOf(Operation))->union(
               self.’
            + ’ownedOperation)->exists(name=\’’ + o.name + ’\’)’);
      } -- add constraint to class
       if (not s.allOwnedElements() -> select(c | c.
          oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c03Name)
              and
          s.oclIsTypeOf(UML2!Class)) {
        thisModule.NewOwnedRule(s, c03Name, c03Expr, ’OCL’);
      }
    }
  }
}
```

Listing A.4: Scenario03.atl

```
-- (c) Stefan Luger 2013
-- Sequence of messages must match sequence of transitions.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario04;
create OUT: UML2 refining IN: UML2;

-- helpers

helper def: getMessagesByClass(cl: UML2!Class): Sequence(UML2!Message) =
  UML2!Message.allInstancesFrom(’INOUT’) -> select(m | thisModule.
      getReceiverLifelineClass(m) = cl);

helper def: getStatemachinesByClass(cl: UML2!Class): Sequence(UML2!
   StateMachine) =
```

```
  UML2!StateMachine.allInstancesFrom('INOUT') -> select(sm | sm.owner =
      cl);

helper def: getReceiverLifelineClass(m: UML2!Message): UML2!Class =
  UML2!Lifeline.allInstancesFrom('INOUT') -> select(l | l.coveredBy ->
      select(i | i.
       oclIsTypeOf(UML2!MessageOccurrenceSpecification)) -> exists(e | e
          = m.
       receiveEvent)) -> first().represents.type;

helper def: getTransitionsByClass(cl: UML2!Class): Sequence(UML2!
    Transitions) =
  UML2!Transition.allInstancesFrom('INOUT') -> select(t | t.owner.owner.
      owner = cl);

helper def: reorderTransitions(st: UML2!Vertex, sm: UML2!StateMachine, l:
    Sequence(UML2!Transition), visited: Sequence(UML2!Vertex)):
    Sequence(UML2!Transition) =
  if visited -> exists(e | e = st) then
    l -> append(UML2!Transition.allInstancesFrom('INOUT') -> select(t2 |
         t2.owner.
         owner = sm and t2.source = st))
  else
    -- append transition to list and recursively call function for
        target state
    UML2!Transition.allInstancesFrom('INOUT') -> select(t1 | t1.owner.
        owner = sm and
         t1.source = st) -- for each source
        -> iterate(i; init: OclAny = OclUndefined | -- call recursively
       thisModule.reorderTransitions(i.target, sm, (l -> append(UML2!
           Transition.
            allInstancesFrom('INOUT') -> select(t2 | t2.owner.owner = sm
                and t2.
            source = st))), visited -> append(i.source)))
  endif;

helper def: traverse(st: UML2!Vertex, i: Integer, t: Sequence(UML2!
    Transition), msgs:
    Sequence(UML2!Messages), tnsns: Sequence(UML2!Transition)): Integer
        =
  if msgs.at(i) = msgs -> last() and t -> exists(tr | tr.name = msgs.at(
      i).name) then
    0
  else
    if not t -> exists(tr | tr.name = msgs.at(i).name) then
      if t = tnsns -> at(1) then
        thisModule.traverse(t -> select(tr | tr.source = st) -> at(1).
            target, i,
             tnsns -> select(tri | t -> select(tr | tr.source = st) ->
                 at(1).
             target = tri -> at(1).source) -> flatten(), msgs, tnsns)
      else
        i
      endif
    else
      thisModule.traverse(t -> select(tr | tr.name = msgs.at(i).name) ->
           at(1).
           target, (i + 1), tnsns -> select(tri | tri -> exists(e | e.
               source =
           (t -> select(tr | tr.name = msgs.at(i).name) -> at(1).target)
               )) ->
           flatten(), msgs, tnsns)
    endif
  endif;

-- new comment constructor alternative
rule NewComment (owner: UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return operation
```

```
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l: String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}

-- for each lifeline, get sequence of messages.
-- for each statemachine representing that lifeline, check wether order of transitions
-- match order of messages
rule Class {
  from
    s: UML2!Class (
      s.oclIsTypeOf(UML2!Class)
    )
  using {
    c04Name: String = 'Sequence of messages must match sequence of transitions.';
    c04Expr: String = '';
    c04Owner: UML2!Class = OclUndefined;
    c04Messages: Sequence(UML2!Message) = thisModule.getMessagesByClass(s); --
         -- ordered already

    c04StateMachines: Sequence(UML2!StateMachine) = thisModule.
         getStatemachinesByClass(s);
    c04Transitions: Sequence(UML2!Transition) = Sequence{};
    c04Start: UML2!Vertex = OclUndefined;
    c04ConstraintViolated: Integer = 0;
  }
  to
    t: UML2!Class (
      -- keep class properties
    )
  do {
    -- reorder transitions
    for (sm in c04StateMachines) {
      c04Transitions <- UML2!Transition.allInstancesFrom('INOUT') -> select(t | t.
           owner.owner = sm);

      c04Start <- let ps: Sequence(UML2!Vertex) =
           UML2!Pseudostate.allInstancesFrom('INOUT').asSequence()
         in
           if ps = Sequence{} then
             OclUndefined
           else
             ps -> select(st | st.owner.owner = sm) -> at(1)
           endif;

          -- from initial state:
      -- look up initial state in transitions as source -> write transition ->
      -- get target, repeat
      -- until all transitions were written into the new ordered sequence of
      -- transitions

```

```
      if (not UML2!Pseudostate.allInstancesFrom('INOUT') -> isEmpty()) {
        c04Transitions <- thisModule.reorderTransitions(c04Start, sm,
            Sequence{},
             Sequence{});

        c04ConstraintViolated <- thisModule.traverse(c04Start, 1,
            c04Transitions
             -> at(1), c04Messages, c04Transitions);

        c04Owner <- sm;
        if (not c04Owner -> allOwnedElements() -> select(c | c.
          oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c04Name)
              and s.
          oclIsTypeOf(UML2!Class) and s.oclIsTypeOf(UML2!Class)) {
          thisModule.NewOwnedRule(c04Owner, c04Name, c04Expr, 'OCL');
        }
        if (c04ConstraintViolated <> 0) {
                  -- add comment stating violation
          thisModule.NewComment(c04Owner, 'Constraint violated at
              message: '.
              concat(c04Messages -> at(c04ConstraintViolated).name)) ->
              debug('CONSTRAINT VIOLATED');
        }

        c04ConstraintViolated = 0; -- reset violation
      }
    }
  }
}
```

Listing A.5: Scenario04.atl

```
-- (c) Stefan Luger 2013
-- A message between two lifelines guarantees an association between the
    two corresponding classes.
-- If an association exists in the opposite direction, the right
    association will still be added, but the former association wont be
    removed.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML

module Scenario05;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: classCons : Sequence(UML2!Constraint) = Sequence{};

helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getReceiverLifelineClass(m: UML2!Message): UML2!Class =
  UML2!Lifeline.allInstancesFrom('INOUT') -> select(l | l.coveredBy ->
      select(i | i.
       oclIsTypeOf(UML2!MessageOccurrenceSpecification)) -> exists(e | e
           = m.
       receiveEvent)) -> first().represents.type;

helper def: getMessagesByClass(cl: UML2!Class): Sequence(UML2!Message) =
  UML2!Message.allInstancesFrom('INOUT') -> select(m | thisModule.
      getReceiverLifelineClass(m) =
       cl);

helper def: getMessageLifelineBySendEvent(snd: UML2!
    MessageOccurrenceSpecification): Sequence(UML2!Lifeline) =
  UML2!Lifeline.allInstancesFrom('INOUT')->select(ll | ll.coveredBy->
      exists(os | os = snd));

-- new comment constructor alternative
rule NewComment (owner: UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
```

```
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return operation
  }
}

-- new operation constructor alternative
rule NewAssociation (aStr: String, cStr: String){
  using {
    a: UML2!Association = UML2!Association.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    a.name <- aStr -> debug('ADD association');
    a.ownedComment <- Sequence{}.append(c);
    a; -- return operation
  }
}

-- new operation constructor alternative
rule NewOperation (oStr: String, cStr: String){
  using {
    o: UML2!Operation = UML2!Operation.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    o.name <- oStr -> debug('ADD operation');
    o.ownedComment <- Sequence{}.append(c);
    o; -- return operation
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
    String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
       INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    --owner.ownedRule <- owner.ownedRule->asSequence().append(c);
    c; -- return constraint
  }
}

rule ClassOwnedAttributeAssociation (rcv: UML2!Class, snd: UML2!Class,
   cStr: String){
  using {
    p: UML2!Property = UML2!Property.newInstanceIn('INOUT')->debug('ADD
       association');
    list: OclAny = OclUndefined;
  }
  do {
    p.name <- rcv.name.toLower();
    p.type <- snd;
    p.association <- thisModule.Association(rcv, snd);
    p.ownedComment <- p.ownedComment->append(thisModule.NewComment(p,
       cStr));
    p.lowerValue <- thisModule.LiteralInteger(1);
    p.upperValue <- thisModule.LiteralUnlimitedNatural(1);
    p;
  }
}

```

```
rule Association (rcv: UML2!Class, snd: UML2!Class){
  using {
    a: UML2!Association = UML2!Association.newInstanceIn('INOUT');
  }
  do {
    a.name <- rcv.name.toLower();
    a.ownedEnd <- Sequence{thisModule.AssociationOwnedEnd(rcv, snd)};
    thisModule.getModel().packagedElement <- thisModule.getModel().packagedElement ->
         append(a);
    a;
  }
}

rule AssociationOwnedEnd (rcv: UML2!Class, snd: UML2!Class){
  using {
    p: UML2!Property = UML2!Property.newInstanceIn('INOUT');
  }
  do {
    p.name <- snd.name;
    p.type <- snd;
    p.lowerValue <- thisModule.LiteralInteger(1);
    p.upperValue <- thisModule.LiteralUnlimitedNatural(1);
    p;
  }
}

rule LiteralInteger (v: Integer) {
  using {
    i: UML2!LiteralInteger = UML2!LiteralInteger.newInstanceIn('INOUT');
  }
  do {
    i.value <- v;
    i;
  }
}

rule LiteralUnlimitedNatural (v: Integer) {
  using {
    i: UML2!LiteralUnlimitedNatural = UML2!LiteralUnlimitedNatural.newInstanceIn('INOUT');
  }
  do {
    i.value <- v;
    i;
  }
}

rule Class {
  from
    s: UML2!Class
  using {
    c05Name: String = '';
    c05Expr: String = '';
    c05Elements: Sequence(UML2!Message) = thisModule.getMessagesByClass(s);
    c05Owner: UML2!Class = s;
    asso: UML2!Association = OclUndefined;
    rcvClass: UML2!Class = s;
    sndClass: Sequence(UML2!Class) = Sequence{};
    ll: Sequence(UML2!Lifeline) = Sequence{};
    assoNav: UML2!Property = OclUndefined;
  }
  to
    t: UML2!Class (
      -- keep class properties
    )
  do {
    for (m in c05Elements) {
      ll <- thisModule.getMessageLifelineBySendEvent(m.sendEvent);
      if (not ll->isEmpty() and not sndClass->exists(e | e = ll->first().represents.type)) {
        sndClass <- sndClass->append(ll->first().represents.type);
      }
```

```
    }

    for (snd in sndClass) {
      -- if asso doesn't exist for snd class, create it
      if (not snd.ownedAttribute->exists(a | a.type = rcvClass)) {
        -- if asso exists in the opposite direction
        if (rcvClass.ownedAttribute->exists(a | a.type = snd)) {
          assoNav <- UML2!Association.allInstancesFrom('INOUT')->select
              (a | a = rcvClass.ownedAttribute->select(a | a.type = snd)
              ->at(1).association)->at(1);
          assoNav.navigableOwnedEnd <- assoNav.ownedElement->debug('
              EDIT navigable <- true');
        } else {
          asso <- thisModule.ClassOwnedAttributeAssociation(rcvClass,
              snd, c05Name);
          snd.ownedAttribute <- snd.ownedAttribute->append(asso);
        }
      }
    }

    for (m in c05Elements) {
      -- add constraint to class
      c05Name <- 'A message ' + m.name + ' between two lifelines
          guarantees an association between the two corresponding
          classes.';
      c05Expr <- 'let l1: Lifeline = Message.allInstances()->select(name
           = \'' + m.name + '\' ).receiveEvent.oclAsType(
          MessageOccurrenceSpecification).covered in' +
      ' let l2: Lifeline = Message.allInstances()->select(name = \'' + m
          .name + '\' ).sendEvent.oclAsType(
          MessageOccurrenceSpecification).covered in' +
      ' let a: Property = l2.represents.type.ownedAttribute in' +
      ' a.oclAsSequence()->notEmpty() and a.type = l1.represents.type';
      if (not c05Owner -> allOwnedElements() -> select(c | c.
      oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c05Name) and
          s.oclIsTypeOf(UML2!Class)) {
        thisModule.classCons <- thisModule.classCons->append(thisModule.
            NewOwnedRule(c05Owner, c05Name, c05Expr, 'OCL'));
      }
    }
  }
}

endpoint rule AppendMultipleConstraints () {
  do {
    for (c in thisModule.classCons) {
      c.constrainedElement->at(1).ownedRule <- c.constrainedElement->at
          (1).ownedRule->append(c);
    }
  }
}
```

Listing A.6: Scenario05.atl

```
-- (c) Stefan Luger 2013
-- Statemachine must be assigned to its corresponding class.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario06;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getStateMachines(): Sequence(UML2!StateMachine) =
  UML2!StateMachine.allInstancesFrom('INOUT');

helper def: getClassByStatemachine(sm: UML2!StateMachine): UML2!Class =
```

```
  sm.owner;

-- new comment constructor alternative
rule NewComment (owner: UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return operation
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
     String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
        INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}

rule StateMachine {
  from
    s: UML2!StateMachine
  using {
    c06Name: String = 'Statemachine must be assigned to its
        corresponding class.';
    c06Expr: String = 'self.owner.oclIsTypeOf(Class)';
    c06Owner: UML2!Class = s;
    c06Elements: Sequence(UML2!Statemachine) = thisModule.
        getStateMachines();
  }
  to
    t: UML2!StateMachine (
      -- keep StateMachine properties
    )
  do {
    -- add comment
--  if (not s.ownedComment->exists(c | c.body = c06Name) and not
    -- s.owner.oclIsTypeOf(UML2!Class)) {
--    thisModule.NewComment(s, c06Name);
--  }

    -- add constraint
    if (not s.allOwnedElements() -> select(c | c.oclIsTypeOf(UML2!
        Constraint)) ->
         exists(c | c.name = c06Name)) {
         thisModule.NewOwnedRule(s, c06Name, c06Expr, 'OCL');
    }
  }
}
```

Listing A.7: Scenario06.atl

```
-- (c) Stefan Luger 2013
-- Statechart diagram must have an initial pseudostate.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML

module Scenario07;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

-- new comment constructor alternative
rule NewComment (owner : UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return operation
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l: String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}

-- new pseudostate constructor alternative
rule NewPseudostate (psStr: String, cStr: String, owner: UML2!Region){
  using {
    ps: UML2!Pseudostate = UML2!Pseudostate.newInstanceIn('INOUT');
    c1: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
    c2: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
    t: UML2!Transition = UML2!Transition.newInstanceIn('INOUT');
  }
  do{
    c1.body <- psStr;
    c2.body <- psStr;
    ps->debug('ps');
    ps.name <- '' -> concat(psStr) -> debug('ADD message');
    ps.ownedComment <- Sequence{}.append(c1)->debug('new comment');
    ps.container <- owner;
    t.container <- owner;
    t.name <- psStr;
    t.source <- ps;
    t.ownedComment <- Sequence{}.append(c2);
    ps; -- return pseudostate
  }
}

rule Region {
  from
    s: UML2!Region (not s.oclIsTypeOf(UML2!Interaction) and not s.oclIsTypeOf(UML2!Class))
  using {
    c07Name: String = 'Statechart region diagram must have an initial
```

```
        pseudostate.';
    c07Expr: String = 'self.ownedMember->select(oclIsTypeOf(Region)).
        ownedMember->exists(oclIsTypeOf(Pseudostate))';
  }
  to
    t: UML2!Region (
      -- keep region properties
    )
  do {
    -- add pseudostate
    if (not s.allOwnedElements()->exists(is | is.oclIsKindOf(UML2!
        Pseudostate))) {
      thisModule.NewPseudostate(c07Name, c07Expr, s);
    }

    -- add constraint
    if (not s.allOwnedElements() -> select(c | c.oclIsTypeOf(UML2!
        Constraint)) -> exists(c | c.name = c07Name) and s.oclIsTypeOf(
        UML2!Region)) {
      thisModule.NewOwnedRule(s, c07Name, c07Expr, 'OCL');
    }
  }
}
```

Listing A.8: Scenario07.atl

```
-- (c) Stefan Luger 2013
-- For each association, the corresponding message must exist.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario08;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

-- only one sequence diagram per model may exist
helper def: getInteraction(): UML2!Interaction =
  UML2!Interaction.allInstancesFrom('INOUT') -> at(1);

-- new comment constructor alternative
rule NewComment (owner: UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return comment
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l:
    String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
        INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
```

```
  }
}

-- new message constructor alternative
rule NewMessage (mStr: String, cStr: String, rcv: UML2!Lifeline, snd: UML2!Lifeline,
    owner: UML2!Interaction){
  using {
    m: UML2!Message = UML2!Message.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    m.name <- '' -> concat(mStr) -> debug('ADD message');
      --m.messageSort <- 'asynchCall';
    m.ownedComment <- Sequence{}.append(c);
    m.interaction <- owner;
    m; -- return message
  }
}

-- new messageoccurencespecification constructor alternative
rule NewMessageOccurenceSpecification (mStr: String, ll: UML2!Lifeline, m : UML2!Message,
    owner: UML2!Interaction) {
  using {
    mos: UML2!MessageOccurrenceSpecification = UML2!MessageOccurrenceSpecification.
        newInstanceIn('INOUT');
  }
  do{
    mos.name <- mStr -> debug('ADD message occurence specification');
    mos.covered <- Sequence{ll};
    mos.message <- m;
    mos.enclosingInteraction <- owner;
    mos; -- return mos
  }
}

rule Association {
  from
    s: UML2!Association (
      not s.allOwnedElements() -> select(c | c.oclIsTypeOf(UML2!Constraint)) ->
          exists(c | c.name = 'For the association ' + s.name + ', a message' +
          ' must exist.')
    )
  using {
    c08Name: String = 'For the association ' + s.name + ', the corresponding message'
        + ' must exist.';
    c08Expr: String = 'let snd: Lifeline = Lifeline.allInstances()->select(represents.'
        + '.type = self.memberEnd->at(1).type) in ' + 'let rcv:' + ''
        + ' Lifeline = Lifeline.allInstances()->select(l | l.represents.type'
        + ' = self.memberEnd->at(2).type) in ' + 'not Message.' +
        'allInstances()->exists(receiveEvent = rcv)';
    c08Snd: Sequence(UML2!Lifeline) = UML2!Lifeline.allInstancesFrom('INOUT') ->
        select(l | l.represents.type = s.memberEnd -> at(1).type);
    c08Rcv: Sequence(UML2!Lifeline) = UML2!Lifeline.allInstancesFrom('INOUT') ->
        select(l | l.represents.type = s.memberEnd -> at(2).type);
    c08MsgName: String = s.name;
    c08SndEvent: UML2!MessageOccurrenceSpecification = OclUndefined;
    c08RcvEvent: UML2!MessageOccurrenceSpecification = OclUndefined;
    c08Msg: UML2!Message = OclUndefined;
  }
  to
    t: UML2!Association (
      -- keep class properties

```

```
      -- add constraint
      ownedRule <- s.ownedRule -> append(thisModule.NewOwnedRule(s, c08Name,
          c08Expr, 'OCL'))
    )
  do {
    -- a lifeline for association member end must exist
    if (not c08Snd -> isEmpty() and not c08Rcv -> isEmpty()) {
        -- if no message exists for the receiver lifeline, add a new one
      if (UML2!Message.allInstancesFrom('INOUT') -> select(m | m.receiveEvent <>
          OclUndefined and m.sendEvent <> OclUndefined and m.receiveEvent.
          covered = c08Rcv and m.sendEvent.covered = c08Snd) -> isEmpty()) {

            -- create new elements
        c08Msg <- thisModule.NewMessage(c08MsgName, c08Name, c08Rcv -> at(1),
            c08Snd -> at(1), thisModule.getInteraction());
        c08RcvEvent <- thisModule.NewMessageOccurenceSpecification(c08MsgName.
            concat('_Send'), c08Rcv -> at(1), c08Msg, thisModule.
            getInteraction());
        c08SndEvent <- thisModule.NewMessageOccurenceSpecification(c08MsgName.
            concat('_Receive'), c08Snd -> at(1), c08Msg, thisModule.
            getInteraction());
        c08Msg.receiveEvent <- c08RcvEvent;
        c08Msg.sendEvent <- c08SndEvent;
      }
    }
  }
}
```

Listing A.9: Scenario08.atl

```
-- (c) Stefan Luger 2013
-- Activity must be represented by an operation.
--
-- @atlcompiler emftvm
-- @nsURI UML2=http://www.eclipse.org/uml2/4.0.0/UML


module Scenario09;
create OUT: UML2 refining IN: UML2;

-- only one model may exist per file
helper def: getModel(): UML2!Model =
  UML2!Model.allInstancesFrom('INOUT').first();

helper def: getActivitysByClass(cl: UML2!Class): Sequence(UML2!Activity) =
  cl.allOwnedElements() -> select(a | a.oclIsTypeOf(UML2!Activity));

-- new comment constructor alternative
rule NewComment (owner: UML2!Element, cStr: String){
  using {
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    owner.ownedComment <- Sequence{}.append(c);
    c; -- return comment
  }
}

-- new constraint constructor alternative
rule NewOwnedRule (owner: UML2!Element, ruleName: String, exp: String, l: String) {
  using {
    c: UML2!Constraint = UML2!Constraint.newInstanceIn('INOUT');
```

```
    oe: UML2!OpaqueExpression = UML2!OpaqueExpression.newInstanceIn('
        INOUT');
  }
  do {
    oe.language <- oe.language -> append(l);
    oe.body <- oe.body -> append(exp);
    c.name <- ruleName -> debug('ADD ownedRule');
    c.constrainedElement <- c.constrainedElement -> append(owner);
    c.specification <- oe;
    owner.ownedRule <- owner.ownedRule -> append(c);
    c; -- return constraint
  }
}

-- new operation constructor alternative
rule NewOperation (oStr: String, cStr: String, owner: UML2!Class){
  using {
    o: UML2!Operation = UML2!Operation.newInstanceIn('INOUT');
    c: UML2!Comment = UML2!Comment.newInstanceIn('INOUT');
  }
  do{
    c.body <- cStr;
    o.name <- oStr -> debug('ADD operation');
    o.ownedComment <- Sequence{}.append(c);
    o.class <- owner;
    o; -- return operation
  }
}

rule Class {
  from
    s: UML2!Class (
      not s.oclIsTypeOf(UML2!Interaction) and not s.oclIsTypeOf(UML2!
          StateMachine)
    )
  using {
    c09Name: String = 'Activity must be represented by an operation.';
    c09Expr: String = '';
    c09Owner: UML2!Class = s;
    c09Elements: Sequence(UML2!Message) = OclUndefined;
    c09Activities: Sequence(UML2!Activity) = thisModule.
        getActivitysByClass(s) ->
         select(a | not s.ownedOperation -> exists(o | o.name = a.name));
    c09Ops: Sequence(UML2!Operations) = Sequence{};
  }
  to
    t: UML2!Class (
      -- keep class properties
    )
  do {
    for (m in c09Activities) {
         -- when there is no super class, add operation to class
         if (not s.allOwnedElements() -> exists(g | g.
              oclIsTypeOf(UML2!Generalization))) {
           c09Ops -> append(thisModule.NewOperation(m.name, c09Name, s))
               ;
         }
         -- otherwise add operation to model, in case it doesn't exist
             yet
         else if (UML2!Operation -> allInstancesFrom('INOUT') -> select(o
              | o.
              owner = OclUndefined and o.ownedComment -> exists(oc | oc.
                  body =
              c09Name)) -> isEmpty()) {
           thisModule.NewOperation(m.name, c09Name, s);
         }
       } -- get all messages for constraint expression
      c09Elements <- thisModule.getActivitysByClass(s);

    -- add constraint
    -- for each Activity, build constraint
    if (c09Elements -> size() > 0) {
       c09Expr <- 'self.inheritedMember->select(oclIsTypeOf(Operation))->
          union(self.'
```

```
          + 'ownedOperation)->exists(name=\'' + c09Elements.first().
             name +
          '\')';

      c09Elements <- c09Elements -> subSequence(2, c09Elements -> size()
         );
      for (o in c09Elements) {
        c09Expr <- c09Expr.concat(' and self.' +
            'inheritedMember->select(oclIsTypeOf(Operation))->union(
               self.'
            + 'ownedOperation)->exists(name=\'' + o.name + '\')');
      } -- add constraint to class
       if (not s.allOwnedElements() -> select(c | c.
          oclIsTypeOf(UML2!Constraint)) -> exists(c | c.name = c09Name)
              and
          s.oclIsTypeOf(UML2!Class)) {
        thisModule.NewOwnedRule(s, c09Name, c09Expr, 'OCL');
      }
    }
  }
}
```

Listing A.10: Scenario09.atl

A.3 Programmatical Launch

```
package at.jku.uml2uml.launcher;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Collections;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.m2m.atl.core.ATLCoreException;
import org.eclipse.m2m.atl.emftvm.EmftvmFactory;
import org.eclipse.m2m.atl.emftvm.ExecEnv;
import org.eclipse.m2m.atl.emftvm.Metamodel;
import org.eclipse.m2m.atl.emftvm.Model;
import org.eclipse.m2m.atl.emftvm.impl.resource.EMFTVMResourceFactoryImpl
   ;
import org.eclipse.m2m.atl.emftvm.util.DefaultModuleResolver;
import org.eclipse.m2m.atl.emftvm.util.ModuleResolver;
import org.eclipse.m2m.atl.emftvm.util.TimingData;
import org.eclipse.uml2.uml.UMLPackage;
import org.eclipse.uml2.uml.internal.resource.UMLResourceFactoryImpl;

/*
 * (c) Stefan Luger 2013
 * ATL EMFTVM programmatical launch configuration launcher class.
 */
public class EMFTVMLauncher {
 private String metaModelName, sourceModelName, targetModelName,
    sourceTargetModelName;
 private String metaModelPath, sourceModelPath, targetModelPath,
    sourceTargetModelPath;
 private String moduleName, modulePath;
 private Metamodel metaModel;
 private Model sourceModel, targetModel, sourceTargetModel;

 ResourceSet emftvmRs;
 ResourceSet umlRs;
 ExecEnv env;
 ModuleResolver mr;
 TimingData td;

 /*
```

```
 * the constructor provides all necessary transformation settings
 */
public EMFTVMLauncher(String metaModelName, String sourceModelName,
   String targetModelName, String sourceTargetModelName,
   String metaModelPath, String sourceModelPath,
   String targetModelPath, String sourceTargetModelPath,
   String moduleName, String modulePath) {
  // initialize UML resource
  initUMLResource();
  initEMFTVMResource();

  // initialize execution environment
  initExecutionEnvironment();

  // initialize model names and file paths
  initTransformation(metaModelName, sourceModelName, targetModelName,
     sourceTargetModelName, metaModelPath, sourceModelPath,
     targetModelPath, sourceTargetModelPath, moduleName, modulePath);

  // instantiate EMFTVM related objects
  mr = new DefaultModuleResolver(this.modulePath, new ResourceSetImpl())
      ;
  td = new TimingData();
}

/*
 * initialize UML resource
 */
private void initUMLResource() {
  Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
     UMLPackage.eNS_URI, UMLPackage.eINSTANCE);
  this.umlRs = new ResourceSetImpl();
  this.umlRs.getResourceFactoryRegistry().getExtensionToFactoryMap()
     .put("uml", new UMLResourceFactoryImpl());
}

/*
 * initialize EMFTVM resource
 */
private void initEMFTVMResource() {
  Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
     "emftvm", new EMFTVMResourceFactoryImpl());
  this.emftvmRs = new ResourceSetImpl();
  this.emftvmRs.getResourceFactoryRegistry().getExtensionToFactoryMap()
     .put("emftvm", new EMFTVMResourceFactoryImpl());
}

/*
 * initialize execution environment
 */
private void initExecutionEnvironment() {
  env = EmftvmFactory.eINSTANCE.createExecEnv();
}

/*
 * initialize model names and file paths
 */
private void initTransformation(String metaModelName,
   String sourceModelName, String targetModelName,
   String sourceTargetModelName, String metaModelPath,
   String sourceModelPath, String targetModelPath,
   String sourceTargetModelPath, String moduleName, String modulePath)
       {
  this.metaModelName = metaModelName;
  this.sourceModelName = sourceModelName;
  this.targetModelName = targetModelName;
  this.sourceTargetModelName = sourceTargetModelName;
  this.metaModelPath = metaModelPath;
  this.sourceModelPath = sourceModelPath;
  this.targetModelPath = targetModelPath;
  this.sourceTargetModelPath = sourceTargetModelPath;
  this.moduleName = moduleName;
  this.modulePath = modulePath;
}
```

```

  /*
   * load/inject models
   */
  private void loadModels() throws FileNotFoundException {
    // load meta model
    metaModel = EmftvmFactory.eINSTANCE.createMetamodel();
    metaModel.setResource(umlRs.getResource(URI.createURI(metaModelPath),
        true));
    env.registerMetaModel(metaModelName, metaModel);

    // load source model
    // sourceModel = EmftvmFactory.eINSTANCE.createModel();
    // sourceModel.setResource(umlRs.getResource(
    // URI.createURI(sourceModelPath), true));
    // env.registerInputModel(sourceModelName, sourceModel);

    // load target model
    if (targetModelPath != "") {
      targetModel = EmftvmFactory.eINSTANCE.createModel();
      targetModel.setResource(umlRs.createResource(URI
          .createFileURI(targetModelPath)));
      env.registerOutputModel(targetModelName, targetModel);
    }
    // load optional combined source and target model
    sourceTargetModel = EmftvmFactory.eINSTANCE.createModel();
    sourceTargetModel.setResource(umlRs.getResource(
        URI.createURI(sourceTargetModelPath), true));
    env.registerInOutModel(sourceTargetModelName, sourceTargetModel);

    env.loadModule(mr, moduleName);
    td.finishLoading();
  }

  /*
   * save models
   */
  private void saveModels() throws ATLCoreException {
    try {
      // targetModel.getResource().save(Collections.emptyMap());

      if (targetModelPath != "")
        sourceTargetModel.getResource().setURI(
            URI.createURI(targetModelPath));

      sourceTargetModel.getResource().save(Collections.emptyMap());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /*
   * launch transformation
   */
  public void launch() {
    try {
      loadModels();
      env.run(td);
      td.finish();
      saveModels();
      System.out.println("TEST: model transformation successful ...");
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (ATLCoreException e) {
      e.printStackTrace();
    }
  }
}
```

Listing A.11: EMFTVMLauncher.java

A.4 GUI Launch

```
package at.jku.uml2uml.gui;

import java.io.File;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

import at.jku.uml2uml.launcher.EMFTVMLauncher;

/*
 * (c) Stefan Luger 2013
 * A simple graphical user interface for a more convenient transformation execution.
 * Simply choose from one of the available transformation scenarios and specify the in/out- as well as an optional target model.
 * Filepaths only work for Windows systems! In case of using Unix, you have to change file separators.
 */

public class Window {
  Display display = new Display();
  Shell shell = new Shell(display, SWT.CLOSE | SWT.TITLE | SWT.MIN);
  String workDir = (String) System.getProperty("user.dir").subSequence(0,
      System.getProperty("user.dir").lastIndexOf('\\'));

  public Window() {
    init();
    shell.pack();
    shell.setSize(600, 175);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
    }
    display.dispose();
  }

  private void init() {
    shell.setText("UML2UML Transformation");
    shell.setLayout(new GridLayout(4, false));
    GridData data = new GridData(GridData.FILL_HORIZONTAL);

    // metamodel
    Label labelMM = new Label(shell, SWT.NONE);
    labelMM.setText("Metamodel name:");
    labelMM.setToolTipText("The metamodel used for the transformation.");
    final Text textMM = new Text(shell, SWT.NONE);
    textMM.setText("UML2");
    textMM.setEditable(false);
    textMM.setEnabled(false);
    final Text textMMPath = new Text(shell, SWT.NONE);
    textMMPath.setText("http://www.eclipse.org/uml2/4.0.0/UML");
    textMMPath.setEditable(false);
    textMMPath.setEnabled(false);
    Label labelMMPH = new Label(shell, SWT.NONE);
    labelMMPH.setVisible(false);

    // module
    Label labelModule = new Label(shell, SWT.NONE);
    labelModule.setText("Module name:");
    labelModule
        .setToolTipText("The ATL module file (*.atl) which contains the transformation rules.");
    final Text textModule = new Text(shell, SWT.NONE);
    textModule.setEditable(false);
```

```
textModule.setEnabled(false);
final Text textModulePath = new Text(shell, SWT.NONE);
textModulePath.setLayoutData(data);
textModulePath.setEditable(true);
Button buttonModule = new Button(shell, SWT.PUSH);
buttonModule.setText("Browse");
buttonModule.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    FileDialog dialog = new FileDialog(shell, SWT.NULL);
    String[] ext = { "*.atl" };
    dialog.setFilterExtensions(ext);
    dialog.setFilterPath("../Transformations/inplace/");
    String path = dialog.open();
    if (path != null) {
      File file = new File(path);
      if (file.isFile()) {
        textModulePath.setText(".."
            + file.getParent().substring(workDir.length())
                .replace('\\', '/') + '/');
        textModule.setText(file.getName().substring(0,
            file.getName().lastIndexOf('.')));
      } else
        textModulePath.setText("");
    }
  }
});

// in/out model
Label labelSTM = new Label(shell, SWT.NONE);
labelSTM.setText("In/Out model name:");
labelSTM.setToolTipText("The UML file (*.uml) which will be transformed by the module specified above.");
final Text textSTM = new Text(shell, SWT.NONE);
textSTM.setText("INOUT");
textSTM.setEditable(false);
textSTM.setEnabled(false);
final Text textSTMPath = new Text(shell, SWT.NONE);
// textSTMPath.setText("");
textSTMPath.setLayoutData(data);
textSTMPath.setEditable(true);
Button buttonSTM = new Button(shell, SWT.PUSH);
buttonSTM.setText("Browse");
buttonSTM.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    FileDialog dialog = new FileDialog(shell, SWT.NULL);
    String[] ext = { "*.uml" };
    dialog.setFilterExtensions(ext);
    dialog.setFilterPath("../Models/papyrus/models/");
    String path = dialog.open();
    if (path != null) {
      File file = new File(path);
      if (file.isFile())
        textSTMPath.setText(".."
            + file.getAbsolutePath()
                .substring(workDir.length())
                .replace('\\', '/'));
      else
        textSTMPath.setText("");
    }
  }
});

// target model
Label labelTarget = new Label(shell, SWT.NONE);
labelTarget.setText("Save as*:");
labelTarget
    .setToolTipText("Optionally saving the target model as a different file (*.uml) to prohibit overwriting the In/Out model file.\nIn order to overwrite the In/Out model file, leave no space (\"\").");
final Text textTarget = new Text(shell, SWT.NONE);
textTarget.setEditable(false);
textTarget.setEnabled(false);
final Text textTargetPath = new Text(shell, SWT.NONE);
```

```
    textTargetPath.setLayoutData(data);
    textTargetPath.setEditable(true);
    Button buttonTarget = new Button(shell, SWT.PUSH);
    buttonTarget.setText("Browse");
    buttonTarget.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
        FileDialog dialog = new FileDialog(shell, SWT.NULL);
        String[] ext = { "*.uml" };
        dialog.setFilterExtensions(ext);
        dialog.setFilterPath("../Models/papyrus/models/");
        String path = dialog.open();
        if (path != null) {
          File file = new File(path);
          if (file.isFile()) {
            textTargetPath.setText(".."
                + file.getAbsolutePath()
                    .substring(workDir.length())
                    .replace('\\', '/'));
          } else
            textTargetPath.setText("");
        }
      }
    });

    Button buttonTransform = new Button(shell, SWT.PUSH);
    buttonTransform
        .setBackground(new Color(Display.getCurrent(), 0, 255, 0));
    buttonTransform.setText("Transform");
    buttonTransform.setToolTipText("Press to conduct transformation.");
    buttonTransform.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
        try {
          new EMFTVMLauncher(textMM.getText(), "IN", "OUT", textSTM
              .getText(), textMMPath.getText(), "",
              textTargetPath.getText(), textSTMPath.getText(),
              textModule.getText(), textModulePath.getText())
              .launch();
        } catch (Exception e2) {
          System.err
              .println("ERROR: Make sure the right ATL module and filepaths
                   are specified correctly! "
                  + e2);
        }

      }
    });

    Label labelPH = new Label(shell, SWT.NONE);
    labelPH.setVisible(false);

    Text textInfo = new Text(shell, SWT.NONE);
    textInfo.setEnabled(false);
    textInfo.setText("Transformation debug information is displayed on
         console.");
  }

  public static void main(String[] args) {
    new Window();
  }
}
```

Listing A.12: Window.java

Bibliography

[1] Eclipse Foundation, Inc., "ATL/Concepts," July 2012. [Online]. Available: http://wiki.eclipse.org/ATL/Concepts

[2] Eclipse Foundation, Inc., "ATL/User Guide - The ATL Language," April 2013. [Online]. Available: http://wiki.eclipse.org/ATL/EMFTVM

[3] Eclipse Foundation, Inc., "ATL/EMFTVM," November 2012. [Online]. Available: http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language

[4] D. Schmidt, "Guest Editor's Introduction: Model-driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[5] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-driven Software Development," *Software, IEEE*, vol. 20, no. 5, pp. 42–45, 2003.

[6] J. Bézivin, "Model-driven Engineering: An Emerging Technical Space," in *Generative and Transformational Techniques in Software Engineering*. Springer, 2006, pp. 36–64.

[7] Object Management Group, "OMG Object Constraint Language (OCL)," January 2012. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1/

[8] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like Transformation Language," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 719–720.

[9] F. Jouault, F. Allilaire, Bézivin., and I. Kurtev, "ATL: A Model Transformation Tool," *Science of Computer Programming*, vol. 72, no. 12, pp. 31–39, 2008.

[10] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification," January 2011. [Online]. Available: http://www.omg.org/spec/QVT/1.1/

[11] M. Tisi, S. Martínez, F. Jouault, and J. Cabot, "Refining Models with Rule-based Model Transformations," INRIA, Research Report RR-7582, Mar. 2011.

[12] Object Management Group, "OMG Unified Modeling Language$^{\text{TM}}$(OMG UML), Superstructure," August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure

[13] Object Management Group, "OMG Unified Modeling Language$^{\text{TM}}$(OMG UML), Infrastructure," August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Infrastructure/

[14] International Organization for Standardization, "ISO/IEC 19501:2005," January 2005. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=32620

[15] Object Management Group, "OMG Meta Object Facility (MOF) Core Specification," June 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1

[16] A. Egyed, "UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 793–796.

[17] A. Reder and A. Egyed, "Model/analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 347–348.

[18] Eclipse Foundation, Inc., "ATL/Developer Guide," August 2012. [Online]. Available: http://wiki.eclipse.org/ATL/Developer_Guide

[19] Eclipse Foundation, Inc., "ATL/User Guide - Overview of the Atlas Transformation Language," July 2012. [Online]. Available: http://wiki.eclipse.org/ATL/User_Guide_-_Overview_of_the_Atlas_Transformation_Language

[20] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3, 2003, pp. 1–17.

[21] F. Jouault and I. Kurtev, "On the Architectural Alignment of ATL and QVT," in *Proceedings of the 2006 ACM Symposium on Applied Computing.* ACM, 2006, pp. 1188–1195.

[22] M. Amstel, S. Bosems, I. Kurtev, and L. Ferreira Pires, "Performance in Model Transformations: Experiments with ATL and QVT," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, J. Cabot and E. Visser, Eds. Springer Berlin Heidelberg, 2011, vol. 6707, pp. 198–212.

[23] A. Demuth, R. Lopez-Herrejon, and A. Egyed, "Constraint-driven Modeling through Transformation," in *Proceedings of the 5th international conference on Theory and Practice of Model Transformations*, ser. ICMT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 248–263.

[24] A. Reder and A. Egyed, "Computing Repair Trees for Resolving Inconsistencies in Design Models," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, 2012, pp. 220–229.

[25] J. Bézivin and F. Jouault, "Using ATL for Checking Models," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 69–81, Mar. 2006.

[26] X. Liu, "Identification and Check of Inconsistencies between UML Diagrams," *Journal of Software Engineering and Applications*, vol. 6, no. 3B, pp. 73–77, 2013.

[27] A. Egyed, "Fixing Inconsistencies in UML Design Models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 292–301.

[28] A. Egyed, E. Letier, and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 99–108.

[29] J. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo, "Orchestrating ATL Model Transformations," *Proc. of MtATL*, pp. 34–46, 2009.