Adding a Module Concept to the Model Transformation Language Xtend

Bachelor Thesis Proposal of

Dominik Werle

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Prof. Dr. Walter F. Tichy
Advisor: Dipl.-Inform. Andreas Rentschler
Second advisor: Dipl.-Inform. Max Kramer

Duration: 13.05.2013 – 13.09.2013
## Contents

1 Introduction ................................................. 3  
1.1 Motivation ............................................... 3  

2 Foundations ............................................... 4  
2.1 Maintainability ........................................... 4  
2.2 Model-driven software development ......................... 4  
2.3 Model-to-model transformations ........................... 6  
2.4 Module concept in general .................................. 6  
2.5 Modular programming ....................................... 7  
2.6 Classification of model transformation languages ............. 8  

3 Example for a transformation: CM2CG ......................... 10  
3.1 Introduction ............................................... 10  
3.2 Decomposition into modules .................................. 11  
3.2.1 Source-driven decomposition ............................. 11  
3.2.2 Target-driven decomposition .............................. 11  
3.2.3 Aspect-driven decomposition ............................. 11  

4 Concept ................................................... 14  
4.1 Planned Approach .......................................... 14  
4.1.1 Theory: Module concept for model transformation languages . . 14  
4.1.2 Implementation .......................................... 14  
4.1.2.1 Xtend ............................................... 14  
4.1.2.2 Limitations of Xtend .................................. 15  
4.1.2.3 Planned Approach ..................................... 16  
4.2 Goals of the bachelor thesis ................................ 16  
4.3 Limitations ................................................ 17  
4.4 Related work .............................................. 17  

5 Work plan .................................................. 19  
5.1 Phases .................................................. 19  
5.1.1 Interface characterization .................................. 19  
5.1.2 Implementation .......................................... 19  
5.1.3 Validation .............................................. 19  
5.2 Artefacts ................................................ 20  
5.3 Schedule ............................................... 20  

Bibliography ............................................... 22
1. Introduction

1.1 Motivation

Model transformations are an important element in the field of Model Driven Software Development (MDSE). They can be implemented in a variety of specialized model transformation languages with special constructs for model transformation or in non-specialized programming languages.

Since models and model transformations are more and more thoroughly understood and applied to more complex problems, the complexity of the meta models and model transformations is growing and non-functional aspects that have traditionally surfaced in software engineering are reappearing in the world of model transformations:

- Maintainability
- Change management
- Reusability
- Bad understandability through complexity

Different approaches have been proposed to handle complexity by using external composition and rule-based composition by inheritance or package import. (See: 4.4. Related Work).

However, those approaches usually focus on reusability and imply a high coupling between the different entities since they rely on white-box composition (no explicit interfaces and therefore no information hiding) and there is not yet a concept for proper interface definition and interface based decomposition into modules (modularization) for hiding complexity of module implementation.

In my thesis I will focus on modularization for maintainability and develop an exemplary implementation of a modularization framework for module transformations in Xtend. The system will use a black box approach for the decomposition of transformations.

The expected benefit of this solution is to ease the process of understanding and maintaining transformations. With a modularized transformation, we expect the location of concerns, i.e. identifying places in the code that have to be modified for planned changes, to be much more efficient.
2. Foundations

2.1 Maintainability

Since my thesis focuses on mechanisms to increase maintainability in model transformations it is important to define this term with respect to the field of software engineering and particularly model transformations.

Maintainability is one characteristic of software quality which includes Modularity, Reusability, Analyzability, Changeability, Modification stability, Testability and Compliance [Int]. It describes

"The degree of effectiveness and efficiency with which the product can be modified." [Int]

Maintainability is moving into a closer focus at the time a product has been delivered and must be maintained to manage possible emerging defects in the product or additional requirements that arise over the course of the use of the software (due to environmental or technological changes, or due to a deeper understanding of the problem domain).

It is generally more expensive to fix problems during maintenance than to avoid them with careful planning in the design phase. The ratio in which the cost increases is called the cost-escalation factor. It can be up to 100:1 [BB01].

Upfront design to make the localization of bugs simpler "by confining them to small, well-encapsulated modules" [BB01] can reduce the cost-escalation factor.

Even though maintainability is often associated with the phase when software has already been deployed it is also important during the design and implementation of the software since it makes reading and understanding the code easier and allows developers to locate possible design problems and erroneous code more easily.

With respect to module transformations maintenance tasks arise when the meta-models change, when defects have to be fixed or new functionality has to be introduced.

In my thesis, approaches for general transformation maintainability (especially the introduction of the module system that is the goal of my thesis) are contrasted with approaches used specifically for reuse, since they differ in their aim and realization. One of the goals of my thesis is to elaborate further on this distinction with respect to model transformations.

2.2 Model-driven software development

Model-driven software development (MDSD) is a modern approach to software development in which the development process uses abstraction and simplification through modeling.

In this section, I am going to clarify some MDSD terms which are important in this context. Stachowiak defines a model as
"A formal representation of entities and relationships in the real world (reduction) with a certain correspondence (mapping) for a certain purpose (pragmat-ics)" [Sta73]

In the realm of software development, a model can be defined more specifically. Voelter contrasts a model in MDSD to a UML model [Voe05] a UML model is merely an abstraction of a programming language, but a model in a broader sense models the application domain and represents formalized domain specific knowledge. It consists of at least one concrete syntax (e.g. in textual or graphical form), and an abstract syntax (the meta model). A model also has semantics which are represented in the concrete syntax. Voelter furthermore states, that in MDSD, the semantic of a model is often defined by model transformations to another language or model with well known semantics.

A **meta-model** is a "model of a model", the abstract syntax of a model. For example, the UML class diagramm meta-model can be used to describe a model (the class diagramm) which in turn describes the "real world". A visual example of this process is given in figure 2.1.

![Figure 2.1: meta models examplified with the UML](image)

The abstraction that is used in MDSD is manifold: on one hand, one typically abstracts from an heterogeneous operational environment (hardware, OS), and on the other hand, the abstraction is from the solution domain to the problem domain, so that domain experts can model the software on domain level without getting into software development details. The motivation for this abstraction is an expected reduction in resources and a better understanding of the problem domain.

The goal of MDSD is to automatically generate software from models that are created by the developer and then gradually processed into other models and code by model transformations. An important aspect of MDSD is the notion that models are first-class entities in the software engineering process. This means that they are not used once to
generate code or to understand the design and then archived for documentation purposes (as it is the case in Model-Based Software Engineering), but rather maintained with the same care as actual code and consecutively used to generate new code whenever a change has been made.

2.3 Model-to-model transformations

Model-to-model-transformations are used to transform models from one meta-model to another (or possibly the same).

They are a fundamental part of the MDSD approach because they make "generating lower-level models, and eventually code, from higher-level models" possible [CH06]. Furthermore they allow to keep different models and code synchronous when changes are made to one. They can also be used to extract parts of models for different users which need a specialized view on the model.

![Diagram of model transformations](image)

Figure 2.2: model transformations (adapted from [KWB03])

Figure 2.2 displays the different parts that are relevant to model transformations and how they are related to each other.

2.4 Module concept in general

To resolve the aforementioned issues with maintainability, I will pursue the idea of modularization of model transformations. This can be achieved by decomposing a model transformation into modules (if it already exists and has to be refactored), or designing a complex transformation in a modular way from scratch.

An informal definition of a module is given by

"A module is a capsule containing (definitions of) items. The module draws a strong (syntactic) boundary between items defined inside it and items defined outside in other modules." [Szy92]

This definition does not specifically address the aim of the modularization or how to find the boundaries of the modules. The way in which these boundaries are defined correlates naturally with the aim of decomposition. In my thesis, modularization for maintainability is in special focus. Therefore I will introduce a more concrete definition given by:
2.5. Modular programming

"A mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time" [Par72]

Parnas also contrasts modules with subprograms by the way decomposition happens: modularization is a task of "responsibility assignment" on "system level" which is part of the design process, and not a refactoring task for complexity that arises during the implementation.

Modularization Principles:

**Low coupling and high cohesion:** Coupling is low, if modules communicate with each other as little as possible, cohesion is high, if the elements of a module are strongly related to each other.

**Information hiding:** A module has an explicitly defined interface. It contains the exported items but *hides the internal structure*. Other modules should always be designed against interfaces.

**Separation of concerns:** Modules are defined according to *design decisions*, and encapsulate one design decision if possible.

To make the purpose of a module clear it makes sense to contrast it with other concepts which are related to modules in the sense that they can be used to structure a piece of software but accomplish this goal in other ways and with different intentions.

A **component** is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Therefore a component is used for composing systems and *specially developed for reuse and composition*. It is usually not the result of hierarchical system decomposition, as it is the case with modules.

A **class** is a somewhat different concept than a module, as it can have *multiple instances*. A module usually does not have an instance (or in some sense only one instance) – all its variables are global and there is only one global scope, not one for every instance.

A distinction of modules to components and classes is that modules are usually resolved during *compile time* and "compiled away", whereas the binding that takes place with components and classes is usually dynamic and can be changed during *runtime*.

2.5 Modular programming

Modular programming is the process counterpart to the concept of software modules. In modular programming, the wanted functionality is decomposed into independent modules during the design phase.

Bertrand Meyer identifies five design **criteria** for a modular design [Mey97].

**Decomposability** The system is decomposed into *independent* modules with reduced complexity. There should be low coupling between modules, so that developers can work on them independently.

**Composability** Modules are designed in a way that they can be *reused* in a different context to create new software systems.
Understandability Modules are designed independently. Thus they should be easily understood by software developers without, or at least with minimal, knowledge of the overall system.

Continuity A reasonably small change in the software system will be localized to a few modules and should particularly not change the boundaries of the modules. This also correlates with the software design principle of High Cohesion.

Protection Errors during the operation of the software system are limited to a small number of modules and should not affect the operation of other modules.

Since Decomposability, Understandability, Continuity are directly related to our notion of Maintainability, I aim to design the proposed module system in a way that makes it possible to adhere to those criteria.

2.6 Classification of model transformation languages

Transformation languages can be classified according to capabilities they exhibit with respect to language constructs that can be used to increase maintainability and for decomposition. Table 2.3 gives an overview over various languages and the planned approach.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Planned</th>
<th>Xtend2</th>
<th>Kermeta</th>
<th>QVT-R</th>
<th>QVT-O</th>
<th>ATL</th>
<th>ETL</th>
<th>VIATRA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Import mechanisms</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Rule inheritance</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Rule merging</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Sumperimposition</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Qualified namespaces</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Explicit interfaces</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Traces</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Functions/methods</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Model elements</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>... Information hiding</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

Figure 2.3: Comparing model transformation languages based on their module system

Even though Czarnecki identifies rule organization as a classification category of module transformation languages (cf. 2.4), "Modularity Mechanisms" in this context are explained as follows:

\(^1\)Epsilon Transformation Language
"Some approaches, e.g., QVT, ATL, MTL, and VIATRA, allow packaging rules into modules. A module can import another module to access its content.” [CH06]

In the course of my thesis, I will expand further on the topic of "Modularity Mechanisms" and give a more fine granular description of how they can be used for module transformations. Furthermore, I plan to expand on the distinction between modularity and reuse mechanisms.

Figure 2.4: Rule Organization [CH06]
3. Example for a transformation: CM2CG

When preparing for this bachelor thesis, a sample transformation has been implemented in Xtend and then refactored to a modular form. The transformation will be used as a case study in the course of my thesis. New techniques will be applied to it as they are developed.

This specific transformation originates from an exercise in the practical course "Praktikum Modellgetriebene Software-Entwicklung"\(^1\) in the winter semester 2012/2013. There, it has been implemented in QVT-O, whereas my implementation will be done in Xtend, and later on use the implemented module and tracing system.

It has been chosen as the running example in my thesis because the usual "toy transformations" appeared to be too simple to be decomposed into meaningful modules.

3.1 Introduction

The Component Model (CM) is a modeling language which is used to model a component based architecture. As a typical architectural description language, it can be used to generate code from, or to analyze it subsequently, for example to compare different architectural alternatives. Furthermore, code stubs can be generated from the model as a basis for further implementation work. CM is a simplified version of the Palladio Component Model\(^2\). In CM, the performance relevant aspects of PCM have been left out for simplicity. It consists of three views: Service Effect Specification (SEFF), Repository and System.

The Repository defines interfaces based on method signatures. Then, components can be defined by declaring which interfaces the component is providing or requiring.

The SEFF describes the implementation of a method of an interface on an abstract level with an automaton. It contains calls to other methods of the same component (InternalAction), to methods of other components (ExternalCallAction), probabilistic branching (BranchAction and ProbabilisticBranchTransition) and connects them with transitions. SEFFs are created for every method of a component.

The System describes how the components are wired together by instantiating the components (AssemblyContext) and connecting required and implemented interfaces (AssemblyConnector).

The Call Graph (CG) models a call graph which describes the flow of a program by connecting subroutines in a multi graph.

The CM2CG transformation takes a CM model and an interface and a method name and creates a call graph which results in the call of the method at the system interface. This transformation to the CG model allows the developer to use existing analysis tools which work on the CG model but not on the CM model.

\(^1\)\text{http://sdqweb.ipd.kit.edu/wiki/Praktikum_Modellgetriebene_Software-Entwicklung_WS12/13}\\
\(^2\)\text{http://www.palladio-simulator.com/}
3.2 Decomposition into modules

Lawley proposes three styles for decomposing a transformation [LDGR04]. In this section, I will motivate my composition of the example CM2CG-transformation.

We chose a decomposition that is not explicitly designed for change, reuse or extension. This would mean trying to predict potential change scenarios, e.g. adding another AbstractAction-type. If we designed for this specific change scenario, it would make sense to design an interface that allows easy extension of behavior.

Source-driven decomposition facilitates an easy localization of the modules that have to be changed. For example, data dependencies to AbstractAction are ideally made explicit in the module interface. Since changes are limited to the SEFF-module’s implementation only, they can be done locally without changing other modules.

The way in which a transformation is decomposed is a design decision. Decomposition directly influences how the system behaves for different change scenarios.

Meyer states that when partitioning a system for Decomposability (cf. 2.5) a method suited for the decomposition is a top-down approach (opposed to a bottom-up approach which is better suited for Reusability). [Mey97]

3.2.1 Source-driven decomposition

This is the approach applicable for the example transformation.

In a source-driven approach, a module’s boundaries are chosen according to the different elements in the source model. In this case, System and SEFF are the modules that we chose. The transformation first traverses the System to locate the correct component instance that handles the respective call at the system interface. Then the repository is queried to locate the right SEFF and traverse them in the right order and create the fitting target model (CG) during the traversal.

My decomposition is displayed in figure 3.1 in a schematic module diagram.

3.2.2 Target-driven decomposition

For this approach, the transformation rules would be aligned to the target elements. Since the target model is rather simple in this case, and target-driven decomposition is rather used for “low-level to high-level” [LDGR04] transformations, we didn’t choose this style.

3.2.3 Aspect-driven decomposition

In an aspect-driven decomposition, rules are not aligned to the source or target model, but rather to aspects or "semantic concepts" [LDGR04]. In the case of our simple transformation, the system and SEFF elements would also be identified as the "semantic concepts" that drive the transformation, therefore distinction between source- and aspect-driven is not very clear in our example.
Figure 3.1: module diagram of the CM2CG-transformation
3.2. Decomposition into modules

Figure 3.2: Structure of the CM2CG transformation with proposed cut
4. Concept

4.1 Planned Approach

4.1.1 Theory: Module concept for model transformation languages

In the context of model transformation the interface of a module naturally includes a collection of rules that can be used and have an interface which usually defines input, output or input-output models they operate on. But there are more subtle aspects which have to be considered when defining an explicit interface for transformation rules, namely traces and overall navigability of the models. The questions that arise are:

- Do traces created by rules always have the same visibility as the rule that created them?
- How is the part of the model that is navigated by a rule specified? Traditionally, the whole model can be navigated by a rule which makes it very hard to narrow down the rules that access a certain meta model element in case of maintenance or change.

By exposing the accessed part of the meta model explicitly in the interface of a rule or the transformation module (and also validating if this interface is adhered to, which is beyond the scope of this work), such tasks are expected to become more easily manageable.

- How is the main method (entry point of the transformation) handled? Is it provided in a special interface?

These questions will be addressed in the theoretical part of my thesis and the CM2CG transformation will be conceptually decomposed into modules with interfaces that adhere to the found interface definition.

4.1.2 Implementation

4.1.2.1 Xtend

Xtend\(^1\) is a programming language that is part of the Eclipse project and compiles into Java source code and can thus be run on the Java Virtual Machine. It is implemented on top of Xtext\(^2\).

The following characteristics of Xtend make it a good choice as a base for the concepts that are to be developed in my thesis:

**Tooling**

Xtend is based on Java and Xtext therefore it is tightly integrated into the Eclipse IDE.

---

\(^1\)http://www.eclipse.org/xtend/

\(^2\)http://www.eclipse.org/Xtext/
Extendability
Since Xtend is based on Xtext, it is also easier to build new language concepts into
the language without getting too much into the details of compiler construction.

Model transformation environment
Even though Xtend is not mainly used for model-to-model transformation, there is
already an environment that can be used to load and process EMF models: the
MWE2 workflows with an XMI-Reader. While this is not optimal for language
independent concepts, I believe it is sufficient for the scope of this thesis as there is
a widespread use of EMF.

Compiles to Java
Xtend is directly integrated into Java’s class and type system. Therefore the Java
class concept can be used (e.g. inheritance, interfaces) and extended (e.g. with data
dependencies or a trace concept). Xtend can also benefit from the optimizations of
the Java Virtual Machine and is therefore highly performant.

4.1.2.2 Limitations of Xtend
Xtend offers support for model transformation workflows via the Modelling Workflow
Engine\(^3\) which allows the user to create pipelines of single elements of the transformation
workflow (e.g. XMI-Reader, XMI-Writer). However, one still has to define the components
for executing a transformation by hand. This can be cumbersome and could be simplified
if there was a common main-method definition for transformations.

```java
component = trafo.CM2CGMweComponent {
  // configuration of the CM2CG transformation
  providedRoleToCall = "Provided_IHTTP_DefaultMediaStoreSystem"
  providedOperationToCall = "IHTTPDownload"

  // configuration for the passing of models from one component to another
  compositionSlot = "inputSystem"
  outputSlot = "transformedModel"
}
```

Listing 4.1: including the custom transformation component for the example

Xtend only provides a very basic tracing method: create methods (cf. Listings 4.2, 4.3).
These work as follows: When the method is called, a hash map is checked for the same
arguments. If there already is an entry, it is returned; otherwise, the transformation
method is called. Results and a hash value computed from the arguments are stored in a
hash map.

This approach works for simple transformations but doesn’t allow more complex opera-
tions, namely late resolving. This technique postpones the resolution of a trace to the
end of the transformation. Another limitation is that for resolution you always have to pro-
vide the same attributes to the method. A more general trace for a model element without
providing the method, for example by just providing the target meta model element.

```python
def create net : CgFactory::eINSTANCE.createGraph toNet(
  ComposedProvidingRequiringEntity self, AssemblyContext context,
  ProvidedRole role, Signature operation)
{
  [...]
}
```

Listing 4.2: create methods

\(^3\)http://www.eclipse.org/modeling/emft/?project=mwe
4.2 Goals of the bachelor thesis

The goals of the thesis are:

- Characterize an interface of a module in the context of model transformation: what elements does it include, which elements are hidden behind an interface?
- Re-use vs. maintainability: How do those concepts differ and how do those differences affect the modularization of model transformations?

\[\text{http://www.eclipse.org/Xtext/7languagesDoc.html#guice}\]
• Insight into control flow dependencies and data dependencies in model transformations and their effect on the interface definition and module design

• Insight into the impact of tracing and explicit hiding/exposing of traces through an model interface on the interface definition and module design

• Definition of possible change scenarios and the benefit of a modularized transformation in those scenarios

• Implementation of a declarative API for module interface definition and implementation as well as a tracing facility in Xtend

• Implementation of a transformation engine by using an existing injection framework (Google Guice) for the composition of the modules in Xtend

• Possibly using the gained insights to introduce new language concepts that support transformation modularization and execution in Xtend

4.3 Limitations

The work in the thesis will be focused on modularization for enhanced maintainability. Since there has already been done a lot of research in the field of reuse of model transformations, this area will not be examined further than necessary in the thesis.

Furthermore the problem of language-independent interface design and composition of modules written in different transformation languages is not a part of this thesis. Still, insights won for the Xtend language should be transferable to other languages, as they will be kept as independent as possible.

The thesis will also only consider imperative model transformation languages since they are used more widely than declarative model transformation languages. While the latter are important enough that they would deserve an inclusion into the observations, this inclusion would certainly go beyond the constraints of this thesis.

4.4 Related work

The approaches in the field of model transformation modularization mostly focus on the topic of reuse.

Kurtev discusses how model transformations can be modularized in rule-based model transformation languages. He exemplifies this process with two scenarios. [KvdBJ06]

Wimmer gives an overview over rule inheritance mechanisms in model transformation languages and proposes a taxonomy for the comparison of them. The taxonomy is then applied to ATL, ETL and TGGs. [WKK+11]

Wimmer also proposes generic model-to-model transformations which are loosely coupled to the meta models. This is done by placing generic parts in the meta model concept that are later bound to concrete elements in the meta model. The approach is expected to foster reuse of transformations across heterogenous meta models. [WKR+11]

Cuadrado discusses how to find overlaps between transformations during the creation (developing for reuse), and how to compose the separated transformations into a system (developing with reuse). [SCGM08]

Wagelaar discusses how to create chains of (black-box) transformations by passing models between them with Domain-Specific Modelling Languages. [Had06]
Olsen defines reuse more specifically to the context of model transformations, describes how transformations have to be designed in order to be reusable and characterizes different ways of reusing. [Had06]

Belaunde describes how composition of transformations can be achieved in QVT-O with imperative constructs for coarse-grained composition (essentially "chaining transformations") as well as fine-grained composition for elementary transformation rules. [Had06]
5. Work plan

In this chapter, I am going to describe the planned course of action for the successful completion of the thesis.

5.1 Phases

Work is divided into three phases, where each phase builds upon the phase before. First, the topic will be worked on conceptually. This is followed by an implementation phase and a validation phase.

5.1.1 Interface characterization

During the first part, the groundwork for all further considerations is established. In the course of the preparation for the thesis and the first devotion to related work in the field a course grained understanding of the scope and the necessary considerations to reach a successful interface characterization has already been established.

Expected Increment

- An interface characterization that defines:
  - What is included into the interface and what is hidden behind it?
  - Which elements does the interface consist of? (in form of a meta model)
  - How are traces handled? Is there a need to include them in the interface, if yes why, if not, why not?
- A conceptually modularized version of the CM2CG-transformation

5.1.2 Implementation

In this part, the theoretical observations from phase one are going to be implemented in the Xtend programming language.

Expected Increment

- An API for offering and requesting transformation module interfaces
- An API for registering and requesting traces for a model transformation

5.1.3 Validation

In the last part, results from the implementation phase are applied to the CM2CG sample transformation. The transformation will be structured and decomposed in Xtend with the API. Furthermore change scenarios will be developed and applied, to validate the decomposition of the transformation on the one hand, and the API on the other hand.

All of the implemented functionality will be tested with unit tests in JUnit\(^1\).

\(^1\text{http://junit.org/}\)
5.2 Artefacts

Expected Increment

- Implementation of the CM2CG sample transformation in Xtend with the API
- Change scenarios and a validation of the change scenarios for the CM2CG transformation
  - How many modules have to be changed?
  - How does the localization of change benefit from modules?

In the course of this thesis multiple artefacts will be created:

- A written report
- An implementation of the module and tracing API written in Xtend
- The CM2CG sample transformation, modularized with the new system
- Change scenarios for the CM2CG sample transformation

5.3 Schedule

Figure 5.1 shows the planned schedule for the bachelor thesis.
Figure 5.1: Schedule for the bachelor thesis
Bibliography


