Marten Sijtema

Managing Variability in Model Transformations for Model-driven Product Lines

Extending the ATL model transformation language with variability management capabilities

Thesis for the degree of
Master of Science

(Computer Science, track Software Engineering)

Graduation committee
Dr. I. Kurtev
Dr. H. Sözer
I. Galvão Lourenco da Silva, MSc.

Faculty of Electrical Engineering,
Mathematics and Computer Science

University of Twente, Enschede, The Netherlands

Abstract

Various approaches show that Software Product Lines (SPLs) can be implemented using the Model-Driven Engineering (MDE) concept of successive model refinements. An important aspect of Product Line Engineering (PLE) is the management of variability.

SPLs can build a set of member products which are subject to variability. That is, the member products will have a varying set of features integrated. Since model transformations can be used to instantiate and integrate components of a system, they should also integrate the features.

This thesis proposes a framework to let the model transformation language ATL manage variability. The approach is to extend ATL (called ATL') with a different type of rule called *variability rule*. This new type of rule works in combination with a separate feature model and an annotation model. The semantics is that a variability rule is only called if the corresponding feature is selected in the feature model, *or* the matched source element is annotated with the feature, as deduced from the annotation model.

The feature model and the annotation model form the drivers for the transformation sequence; they are used for the configuration of features. Specifically, the feature model defines the set of selected features, and the annotation model provides a more granular way of specifying feature integration on a per-class basis. The application engineering phase – the product derivation – is then fully automated.

A compiler was written as a higher-order transformation (HOT), to implement the semantics of variability rules. This HOT compiles an ATL' model back to an ATL model. The HOT can be chained in front of a transformation step, and thus be regarded as a pre-processor.

Throughout the thesis, a transformation sequence that can generate a family of web-based, data-centric information systems with basic Create, Read, Update, Delete (CRUD) support, is used as a case study. It is the context for all the experiments and examples regarding variability. The transformation sequence generates member products from an input (meta) model, the domain model. The domain model describes a particular business domain and is an instance of the Ecore meta meta model. Every generated information system will provide means to store and manage data as described in the domain model, and integrate a varying set of features.

Our solution ensured separation of concerns, modularity, and higher maintainability. The variability rules have similar quality characteristics as normal rules, and its semantics follow the ATL philosophy of modular, declarative rules, and implicit execution order.

Acknowledgements

I would like to use this page as an opportunity to thank some people for their help and support.

First and foremost I would like to thank my first supervisor, Ivan Kurtev. My first inspiration for a model-driven product line was during his course on Model-driven Engineering. The idea of building a model-driven product line, and doing 'some research' on it, was OK with Ivan, and he allowed me to do a bottom-up research to see where the limitations were.

It turned out to be variability management, a very interesting topic, and during the process his constructive input really helped me to stay on track. We had a very nice informal way of working, and he helped me out with technical questions very quickly.

He also encouraged me to write a paper about one of the results. The paper got accepted for the MtATL 2010 workshop in Malaga, Spain. So we travelled there, and had a good time.

In Malaga, I spoke with Frédérique Jouault, one of the authors of the ATL language. He helped me out with a lot of questions about language intricacies, and with some implementation issues. So, thanks to him as well.

Thanks, also, to Hasan Sözer and Ismênia Galvão Lourenco da Silva, for taking the time to read and review this thesis, and participating in the committee.

I would also like to thank my parents and the rest of the family for being supportive and interested in my life as a student. They wanted to make sure that I graduated one day, and I am glad that I succeeded in this.

My final thanks goes out to all my friends, and in particular my girlfriend, Irene. They made my life as a student so fun, entertaining and interesting, especially when not at the university campus.

Now, the adventure begins. Starting with building a company around model-driven engineering and software product lines, with my good friend and fellow student Thijs ten Hoeve. I am looking forward to use these results from academic research and put it into practice as an entrepreneur!

Marten Sijtema, September 2010, Enschede

Table of Contents

List of Figures

1 Introduction

1.1 Background

1.1.1 Model-driven Engineering

The classical way for understanding complex real-world problems, is to build a model. Essentially, a model is a simplification of the real-world problem, that is, it is more abstract. Raising the level of abstraction helps understanding the complex world. A model can be anything, from a mathematical equation to a graph-like diagram to a tangible object. In software engineering, the concept of MDE was introduced to continue the trend of raising the level of abstraction [AK03]. MDE facilitates in bridging the gap between software models with a high level of abstraction and program code, which contains all the nitty-gritty details and has a very low level of abstraction. This means that in MDE, models play a direct role in software engineering, having models that are interpretable, and transformable by machine [Ken02].

A statement used for MDE is: "*In MDE, everything is a model*" [Béz05]. This means that every artefact in the MDE process can be regarded (and interpreted) as a model, be it manually or in an automated fashion.

Without MDE, software engineers start out with models, and eventually end up with program code. Programmers would interpret a model manually, and write program code that corresponds to the model, but has more details added – a lower abstraction level. Often, this gap is very large. MDE technologies try to automate the process of lowering the abstraction level, by providing facilities to refine models in a step-wise fashion.

A key MDE concept to achieve this is called *model transformations*. As the term implies, models are step-wise transformed (or refined) from a high abstraction level to a lower one, eventually reaching the abstraction level of program code, or at least a level where it can be executed directly. A model transformation is written in a model transformation language like ATL [JK06], or MediniQVT [IKV].

In recent years, there are different applications of model transformations, other than step-wise refinement. For instance, it is used to transform models from one domain to one of another domain.

With MDE, one has to create (or reuse) a meta model for each abstraction level, to be able to write a model transformation. A model can be developed using a modeling language. A modeling language uses a meta model which specifies the 'grammar' for each model. That is, a meta model defines the structure which all models created with the modeling language follow. We say that if a model conforms to a meta model, it is *an instance of* a meta model.

1.1.2 Product Line Engineering

When building a family of software products, one can apply the concepts of PLE [PBvdL05]. A product family is a set of products that share a common base in terms of functionality or architecture, but differ on certain aspects. In other words, the members of a product family have commonalities and variabilities. PLE provides methodologies to effectively capture and reuse the common parts, and also provide techniques to manage the variable part.

To do so, building a product line consists of two phases: domain engineering and application engineering. In the domain engineering phase, all the common parts (ie. models, modules, etc.) are developed, as well as models that describe the variable parts and their relations. These models, documents and modules are called domain artefacts. The development of domain artefacts is done systematically, with the focus on reusability. This means that developed components have to be generic.

In the application engineering phase, different common parts and variable parts from the domain engineering phase are assembled into one application.

One of the hardest parts is the management of variability. Often a feature model is defined which describes all the possible options that a member of the family can have [BPSP04]. Ideally, the feature model is separate, and used to configure the final product, after which a family member can be derived automatically.

The concepts of PLE and MDE can be used together. The automatic product generation can be quite complex because of feature integration, and MDE technologies have proven to be well-suited for integrating common parts and selected variants [GPA+07, VG07].

1.1.3 Using MDE technology for building a product line

Research work from the past has shown us [GPA+07, VG07] that activities, artefacts and infrastructure of Product-Line Engineering (PLE) can be executed with concepts, technologies and methodologies from Model-driven Engineering (MDE). This thesis describes a (bottum-up) research that starts with a product line built with a proven MDE paradigm: step-wise model refinement. Then, a prominent problem was identified with this approach: variability management was hard.

The product line in this thesis consists of web-based, data-centric information systems, with basic support for the data operations Create, Read, Update, Delete. For instance, a typical member product could be a basic Customer-Relations Management (CRM) system, where customer/relations could be dis-

played, added, updated, or removed. Note that it does not matter what the data is: it could be customers, persons, appointments, inventory, etc. Chapter 3 describes the example case in more detail.

The product line is built with a MDE infrastructure, that is, it is a transformation sequence. The transformation sequence, and its elements (ie. models and meta models) can be regarded as a software platform, which is a common term from PLE. Family products are automatically derived by automated step-wise model refinement. In the initial version of the software platform, one could build a model (a class diagram) containing the elements that should be manageable in the final product (in the CRM case it would contain a Customer and a Relations class for instance). In other words, one should model the domain logic of the eventual system in a class diagram. This model would be the input of the transformation sequence, which step-wise refines it until working code is generated.

1.2 Problem statement

The transformation sequence has a main problem in its inability to integrate variable components into a final product. Specifically, given a fixed input model, there would be one possible derived product. In practice, however, feature configuration is important. For instance whether or not to integrate a search module, what kind of database technology to use, what kind of user interface to use to display the data, etc.

Variability management (with a fixed input model) in the context of our transformation sequence is the main focus of this thesis. We try to solve the problem for the more general concept of step-wise model refinement; given a fixed input model x and a step-wise refining transformation sequence t, a set of possible products S should be derivable. Furthermore, elements (ie. models, transformations, etc.) of t should have a good quality characteristics.

The quality properties of the model transformation definitions should be the following:

- maintainability and ease-of-use, by using declarative code as much as possible, thus avoiding imperative code as much as possible,
- modularity, using modular constructs as much as possible, which should also improve reusability,
- extensibility, a transformation definition should be easily extended with the integration of new features,
- separation of concerns: concern of configuration separated from model transformations.

We will use the popular open-source model transformation language ATL for specifying our model transformations. The following problem statement is addressed:

Problem Statement. *In MDE-based Product-lines, many variable components should be integrated/instantiated by executing model transformations.*

The popular model transformation language ATL is not capable of expressing this whilst conserving the quality properties stated above.

The problem statement leads to the following research question.

Main Research Question. *How to manage variability in ATL model transformation definitions in MDE-based product lines that derive products by step-wise refinement?*

The main research question does not only apply to our transformation sequence, but any step-wise refinement transformation sequence that needs to integrate variable parts. It is assumed, however, that ATL is used as the model transformation language. Furthermore, the running example case is used as a starting point in assessing the practical needs and problems, which means that it is a bottom-up research.

The main research question can be divided into three sub-questions.

RQ1. *What kinds of variability are presented in product members of model-driven product lines?*

The goal is to identify and classify variability that exist in the product members. The eventual product is what counts, thus we use variability needs in products as a starting point. Once, from a 'business' point of view, there is insight in the potential variable requirements between the generated products, the second sub-question emerges.

RQ2. *How can the MDE infrastructure be adapted to manage these types of variability, conserving the quality characteristics from above?*

In other words, what technologies, methodologies, (meta) models or guidelines are needed to manage the variability efficiently? The result needs to be implemented, achieving better variability management in the product line. There will be an evaluation of to what extent this was achieved.

The previous question can be made more specific, by applying it to ATL, the model transformation language. An assessment needs to be done to what extent ATL is suitable for variability management. Limitations of ATL need to be overcome on this subject. This leads to the final research question:

RQ3. *What are the limitations of ATL when it comes to variability management, and how can these be overcome?*

1.3 Research objectives

The main objective is a transformation sequence that is able to manage (relevant) variability efficiently. To do so, the infrastructure is examined, and limitations are identified. However, it is essential that the solution could also be applied in other situations. That is, the solution should be reusable for other transformation sequences that need to manage variability.

1.4 Contributions

We now list the main contributions of this thesis.

- By answering **RQ1**, we created a classification of relevant variability that occurs in model transformations in model-driven product lines. We identified two types of variability: source model dependent, and source model independent variability, which will be explained in Chapter 4
- An enhanced MDE infrastructure was developed, with means for managing variability in an efficient way, answering **RQ2**.
- Answering **RQ3** pinpointed the problem of ATL in its ability to manage variability, and provided a solution for this, in the form of variability rules.
- To accommodate variability rules, we created a feature model editor, an annotation model editor, and a HOT. The HOT interprets variability rules, and implements the semantics. The status of the package is currently a proof-of-concept, but is usable.
- Along with these artefacts, we defined a set of usage guidelines, that is, a recipe for using the infrastructure. This is described in Chapter 6.
- The usage guideline shows that our concept can work in any transformation sequence that involves variability.

1.5 Outline

- Chapter 2 introduces the basic concepts used throughout the thesis. It will explain key concepts within MDE and PLE and some applications of these engineering disciplines.
- Chapter 3 explains the product line example that is used throughout this thesis. Since it is used as context for all the examples, and since it comes from the business, this chapter will provide the context and motivation.
- Chapter 4 explains that variability that occurs in the example product line. It is not a general chapter about variability management, but geared towards the product line of this research. It identifies what types of variability occurs in the product members of the product line.
- Chapter 5 explains the problem with the transformation sequence as it was at the beginning of the research, and provides a solution. The solution is to extend the ATL model transformation language such that it can cope

with the variability types that were identified. Furthermore, it explains what types of extra models (being feature models and annotation models) need to be used in the transformation sequence to be used as parameters in the ATL solution.

- Chapter 6 explains guidelines on how to use the extended ATL and the supporting models.
- Chapter 7 evaluates the solution, by comparing it to other approaches.
- Chapter 8 concludes this research. This chapter answers the research questions.

The next chapter can safely be skipped if one is already familiar with MDE and PLE concepts. The chapter dependencies are shown in Figure 1.1.

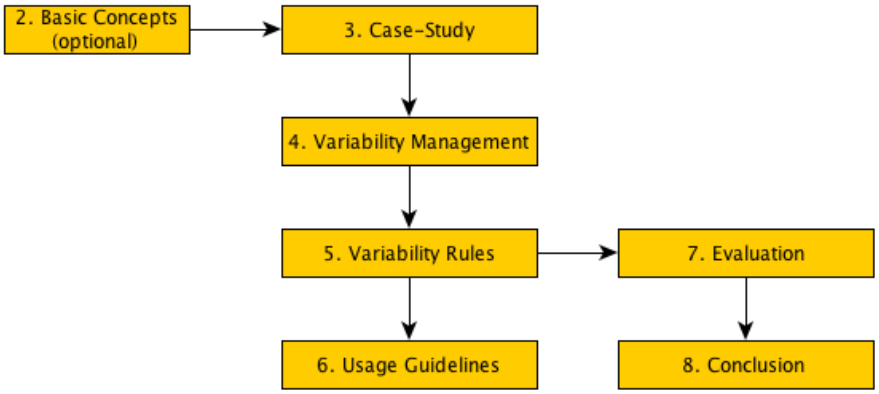

Figure 1.1: Dependencies of the chapters.

2 Basic Concepts

2.1 Introduction

This chapter introduces basic concepts, definitions, and common technologies and applications in the field of Model-driven Engineering and Product-line Engineering. It can be skipped if one is already familiar with them. Also, this chapter is more elaborate than the concepts introduced in the introduction.

Section 2.2 is about model-driven engineering. It discusses models, meta models, meta meta models, model transformations, different frameworks, the XML Metadata Interchange (XMI) format, and a few common application areas of MDE.

Section 2.3 is about PLE. It discusses the philosophy, and the phases product line engineers divide their projects in. Also, a key concept within PLE is discussed: variability management. It shows some approaches. This chapter explains the general concept, more information about variability management in the context of step-wise model refinement using MDE technologies can be found in Chapter 4.

2.2 Model-driven Engineering

The original motivation of model-driven engineering is the notion that there is a gap between the abstraction level of program code and problem domain models. In the classical way of software engineering, the first stages of a project would be to develop requirements of a system, followed by the development of models. Models would give a better understanding of the system that is to be developed, and can be made from different perspectives, and can vary in level of detail. The level of detail, called 'level of abstraction' or 'abstraction level', often is much higher (that is: more abstract or less detailed) than the eventual program code. This 'semantical gap' potentially causes trouble. Software design models are interpreted manually by the programmers who write the eventual

code, and because of the lack of detail, there is a potential problem of the code being inconsistent with its corresponding (set of) model(s).

To bridge this gap, and to overcome the potential inconsistency of models and code, MDE was proposed. A main goal was to be able to connect models to code, or to let code follow from models, in a (semi-)automated fashion. To do so, a paradigm was proposed which based around a fundamental principle: **In MDE, everything is a model**. This is comparable to the object-oriented paradigm, where everything is an object.

2.2.1 Everything is a model: models, meta models, and meta meta models

Having a model as a central artefact, some structure is needed to facilitate the automated processing of models. In programming languages, this would be done by grammars, concrete syntaxes and their corresponding parsers. In MDE, this structure is provided by, indeed, models. Specifically, the structure of every model is expressed in a (graphical or textual) modeling language, and the structure (read: grammar) of this language is described in a model, called the **meta model** [Béz06].

Meta models are, again, all following the structure of a different kind of model: the **meta meta model** [Béz06]. Ideally, there is just *one* meta meta model, thus all meta models obey the same meta meta model. The meta meta model is following its own structure; it is expressed in terms of itself. Schematically, we have the model stack shown in Figure 2.1.

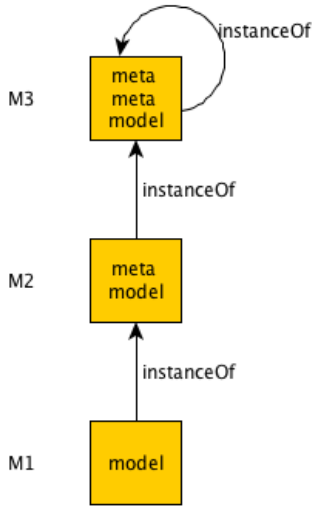

Figure 2.1: Three level model stack used in MDE.

The *instanceOf* relation is an important relation in MDE, and is defined as shown in Definition 1. This definition is, however, nor complete nor formal, it is shown here to provide an intuition.

Definition 1 (*Instance of* relation). A (meta) model M is **an instance of** a meta (meta) model MM if it is created using the modeling language which is based on MM.

Here, the three levels are shown. The stack provides a good structure to create modeling languages and thus models of any domain. Furthermore, its structure allows tools to automatically generate parsers and editors for modeling languages. The following list relates the concepts discussed so far:

- A modeling language allows the creation of models (M1).
- A modeling language follows the structure, or 'grammar', of a meta model (M2).
- All meta models, and thus all modeling languages, follow the structure of a common meta meta model (M3).
- The stack allows tools to (semi)-automate the creation of the infrastructure for a modeling language, consisting of: concrete syntaxes, parsers, editors, and code generators.

To conclude this section, the following definitions are given following the analysis by Kurtev [Kur05].

Definition 2 (Model). *A model is an abstraction of a part of the reality for a specific purpose. A model is expressed in a modeling language.*

Definition 3 (Modeling language). *A modeling language is a well understood (not always formal) language which describes the concepts and their relations in a part of reality.*

Definition 4 (Meta model). *A meta model (at M2) is a model of a modeling language.*

Definition 5 (Meta meta model). *A meta meta model is a model (at M3) of a modeling language, that is, a model to which a meta model conforms to. A meta meta model is expressed in terms of itself, thus is an instance of itself.*

Some attempts have been made that try to formalize the concepts discussed here [Fav04][Béz06]. We will not elaborate on this further.

2.2.2 Implementations of the model stack

There are multiple frameworks that use the stacked structure shown in Figure 2.1. A few of them are mentioned briefly.

XML and BNF from an MDE point of view

The eXtensible Markup Language (XML) [xml] also has a stacked structure. An XML document (M1) is an instance of an XML Schema (M2). An XML Schema in turn is an instance of an XML Meta Schema. The XML Meta Schema is an

instance of itself. Note that XML was not initially invented for the purpose of MDE, but it does follow the stacked structure.

The stacked structure also holds for a normal computer program, written in a programming language. One could observe that the grammar of a programming language is written in Backus-Naur Form (BNF). BNF, in turn, is written in itself. An actual program (M1) is an instance of its grammar (M2) is an instance of BNF (M3).

Model-Driven Architecture, by Object Management Group

The Object Management Group (OMG) proposed a generic concept called Model-Driven Architecture (MDA™). Their meta meta model is called the Meta Object Facility (MOF). MOF 1.4 [MOFa] and MOF 2.0 [MOFb] are the released versions.

The main idea was that MOF would be a meta meta model for the Unified Modeling Language (UML). However, there was a problem with it: many people found it too complex. To reduce this complexity, a subset of MOF was defined, called Essential MOF (EMOF). Unfortunately, this is still not practical enough.

Eclipse Modeling Framework

There was demand for a simpler approach than MOF. Probably the most popular one is called the Eclipse Modeling Framework (EMF) [BBM03]. This approach follows the same concept as MDA, and was geared to practical use. EMFs meta meta model is called ECore. ECore is very similar to EMOF.

Today, there is a large suite of tools based on EMF, and these can all be used from the popular Eclipse IDE. These tools range from code generators, parser generators, concrete syntax specification tools, model-to-model and model-to-text transformation languages, and rich editors complete with outline, syntax highlighting, auto-completion, etc.

We also use the EMF framework, and a range of tools that are implemented around it.

2.2.3 Model transformations

A benefit of the model stack is that models can be transformed into other models. In other words, it allows the transformation of a model which is an instance of meta model A into a model which is an instance of meta model B. This activity is called *model transformation*. Model transformations are expressed in model transformation languages.

Note that a model transformation language is a modeling language. So model transformation languages are also based on a meta model, and every written model transformation can be regarded as an instance of a meta model. Following this principle, it is possible to write a transformation that transforms a model into an instance of a transformation language meta model, meaning that the output of the transformation is a transformation itself. This is called a higher-order transformation (HOT) [TJF+09][TCJ10]. We use a higher-order transformation as a compiler.

2.2.4 Step-wise model refinement

A common application of model transformations is to transform a model with a higher level of abstraction (ie. an instance of a more abstract meta model, *or* an instance of the same meta model, but with less detail), into a model with a lower level of abstraction. One can step-wise refine a model until the abstraction level is low enough to be executed directly, or to be easily converted into program code. This thesis uses this concept of **step-wise model refinement** [BSR03] to generate web-based, data-centric information systems with basic Create, Read, Update, Delete (CRUD) operations.

Such a sequence of model transformation steps is called a **transformation sequence**. In our transformation sequence case, as the next chapter will show, we start off with an abstract model which only models the general concepts of a specific domain. In the transformation steps that follow, details are added (ie. there is more detail in the meta model, and thus in the instances of these meta models, the models) like database classes, user interface classes, etc.

2.2.5 Transforming models from one domain to another

More recently, it proved useful to create model transformations that transform a model from one domain into one from another domain. For instance, to transform models from the database query language MySQL to Oracle's query language. Also attempts have been made to transform object-oriented models into relational database models and vice versa (see for instance [SQL]), giving a model-driven approach for Object-Relational Mapping, comparable with ORM technologies like Hibernate [Hib].

2.2.6 Model injection and extraction

Often it is convenient to have a graphical or textual concrete syntax for modeling languages.

For textual modeling languages, there are for example XText[oAW] and TCS [JBK06]. Both are used in this thesis. XText allows the specification of a grammar enriched with language constructs. From the grammar a meta model is inferred, which conforms to ECore. XText generates a parser and an editor as an Eclipse plugin for the specified syntax.

The same applies for a similar technology called TCS (Textual Concrete Syntax). The difference with this approach is, that one first specifies the meta model (also conforming to Ecore), and then provides a template for the concrete syntax. Like XText, a parser is generated along with an editor.

There is a key concept used by both systems: injection and extraction [JBK06]. Injection is the process of parsing a textual version of a model, into a run-time, object-oriented, in-memory model. The model can be used by transformation engines, code generators, or tools that analyze model properties. An injector is automatically generated by TCS and XText.

The reverse process, having an in-memory model and serializing it to a textual version of the model (in its correct syntax), is called extraction. The extractors are capable of pretty-printing the model, that is, using information from the XText/TCS grammar to indent the code on the correct places.

Observe that the meta model of a modeling language is the main guide in the injection and extraction process. Given the fact that each meta model conforms to the ECore meta meta model, all EMF tools use a common ground, which makes it easy to develop new tools.

To conclude this section, let us look back at the previous one. An injector/extractor pair for the MySQL language and Oracle query language would be needed to convert existing MySQL files into Oracle files. This requires having a meta model conforming to ECore. There are various people that proposed such meta models.

The injector/extractor paradigm can be used to cross the borders of technology domains, and allows to make this technology compatible with the world of MDE.

2.2.7 Domain-specific Languages and MDE

When code generators or interpreters are written that process an injected model, a programming language is born. This can be a general purpose language, or a domain specific language (DSL) [MHS05]. Due to the large set of EMF-based tools dedicated for the creation of DSLs, MDE proves to possess practical ways of creating DSLs.

2.3 Product line Engineering

Although the focus is more on MDE, there is a substantial use of PLE concepts in this thesis. Therefore, we explain the common terminology.

2.3.1 What is a Software Product line?

An SPL is, a set of related software products. As an example, consider Microsoft's Office Suite. It includes MS Excel, MS Word, MS Visio etc. The important characteristic of an SPL is that the different products in it share a common base, that is, they have commonalities. On the other hand, the different members of the product line (also called 'family members') can have specific features not integrated in other members. These two opposite sets are called *commonalities* and *variabilities*.

The fundamental idea of PLE is to capture commonality amongst products, and manage the variability. To do so effectively, the creation of an SPL is done in two phases: the Domain engineering phase, and the Application engineering phase.

2.3.2 Domain Engineering

In this phase, it is investigated what is common amongst products. For these common parts, software artefacts are created. This can range from various design models, requirements, or actual programmed modules. Once this is done, these domain artefacts can be reused. Thus, there is a systematic (although not necesarilly formally specified) way of building domain knowledge and capturing this in *reusable* artefacts.

These artefacts consist of a common reference architecture, which is the most important reusable asset. Furthermore, the different components should be generic, to ease integration in member products.

The variable part, which is inevitably existent in an SPL, has to be managed. The variation points are often modeled in a separate model. This model describes the points on which different member products can differ. Also, it puts constraints in what features are allowed in combination with other features.

2.3.3 Application Engineering

Once the domain artefacts are established, one can create different member products using the domain artefacts. This activity is called application engineering. The composition/integration of different domain artefacts can be done automatically or manually.

Ideally, there are domain artefacts that facilitate the derivation of member product in the application engineering phase. An example would be domain-specific languages (DSLs).

2.3.4 Software Platform

The eventual set of domain artefacts and application engineering facilities in an SPL can be considered a Software Platform. Pohl et al. [PBvdL05] defined a software platform as:

> *"A software platform is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced."*

Observe that it is possible to consider the CRUD information system generator case a Software Platform.

2.3.5 Variability Management

Variability is a term used to denote the different solutions that are available for a certain concern. Or, according to [hs10]:

> "Variability Management is the overall process of defining what is common and what is different across Products in a Product Line (i.e. allowable Product Configurations), as well as managing the set of actual Product Configurations."

Managing the integration of variable components tends to be complex. A good starting point has proven to be to model the variable parts separately, and treat it as an orthogonal concern [PBvdL05]. This means that there is an orthogonal variability model, which contains a notion of all the variable variants, or features from products in the SPL. One way of expressing an Orthogonal Variability Model (OVM) is by the graphical syntax of a feature model [PBvdL05].

Feature Models

Feature models describe the features in a system, and allow for selecting a subset of features according to the selection/configuration rules applied to the features. Typically, in the simpler case, a feature model is a tree, where each subtree is a feature. Feature models supply a few constructs to constraint the possible selection of features:

- mandatory — the feature has to be selected,
- optional — the feature can or cannot be selected,
- alternatives — operates on multiple features, stating that precisely one of them has to be selected, like an xor-operator,
- m-to-n — atleast m and at most n features that are operated on should be selected. Cardinalities denote the value of m and n. If cardinalities are omitted, the default values of m and n are 0 and the number of operands of the m-to-n operator, respectively, which makes the m-to-n operator an inclusive-or operator.

Furthermore, there are two types of links which put additional constraints between features:

- requires — the source feature requires the target feature to be selected, in order for the source feature to be selectable,
- excludes — the source feature, if selected, excludes the target feature from being selected.

Figure 2.2 shows the meta model of a feature model, which is created by us. There are other meta models, but there is not a standard representation of feature models in EMF, thus we generated a tree-based editor from our own meta model using standard tools. This eased the usage of feature models in our model transformations.

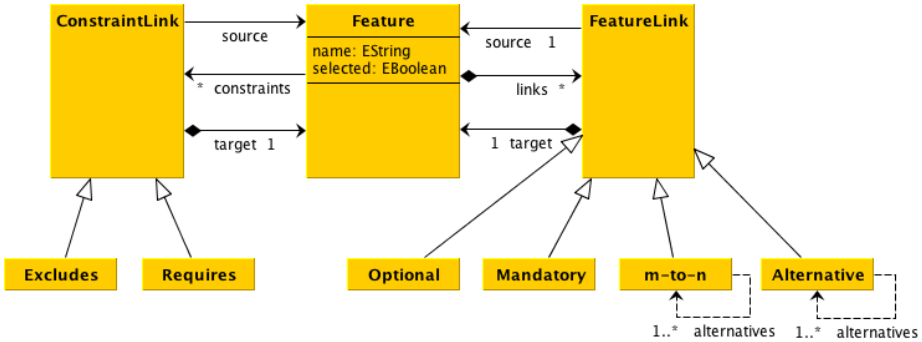

Figure 2.2: Feature model meta model.

Example Feature Model

An example feature model is shown in Figure 2.3. Figure 2.4 shows the legenda. Here, a car is used as an example, which can typically vary on a lot of aspects. One could in this case opt for a diesel engine *xor* a petrol engine. The exclusive-or relationship, that is the alternative relationship is denoted by an open arc grouping. If this grouping arc is filled, it means that one or more of the features from the group are allowed for selection, as is the case with engine add-ons. In some versions of feature models cardinalities are added to specify a minimum and maximum number of selected features in a specific group. Some features are optional (denoted by an edge with an open circle), like the airconditioning. Lastly, a gearbox is mandatory, which is denoted by a solid circle at the end of an edge.

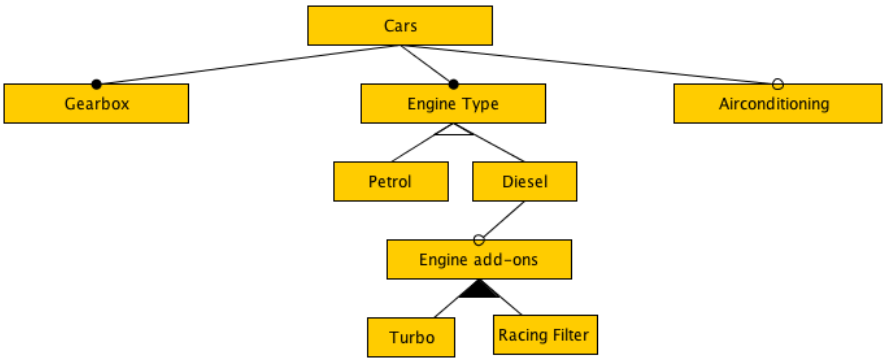

Figure 2.3: Example feature model in the domain of cars.

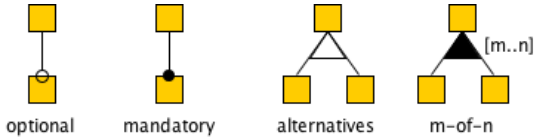

Figure 2.4: Feature model legenda.

It is important to stress the word 'orthogonal'. It means that the variability model is not only separate, but also relates to all domain artefacts and reference architectures. In other words, traces can be made between elements from the OVM to various domain artefacts.

These traces or mappings can be useful for (semi-)automatic feature integration.

Especially in the application engineering phase, where different features have to be integrated into a member of the SPL, efficient variability management is important. The ideal case is to automate the integration and composition of variable and common features into a product.

2.4 Using MDE for PLE

Observe that MDE frameworks like EMF would be a good candidate for helping to develop domain artefacts (ie. models and meta models), because of the well-defined structure of models and meta models. There are researches showing that automatic product derivation from a set of domain artefacts can be done efficiently with MDE technologies like model-to-model transformations and model-to-text transformations. Specifically, the step-wise refinement paradigm proves very powerful [HKW08][ACR].

In our opinion, using MDE for PLE is a good approach. When developing, relating and/or generating domain artefacts, MDE provides exactly the right structure and tools to do so. The M3 level makes sure that everything is coming from the same base, easing the required mappings and transformations that have to be made. MDE technologies are furthermore very well suited for generating DSLs and all the infrastructure like code-generators, editors, and parsers. This is the main motivation for building the CRUD information system product line from this thesis using step-wise model refinement.

2.5 Conclusions

This section showed the basic concepts in MDE and PLE. For MDE, It introduced the model stack, meta models, models, and model transformations. For PLE, the key concepts of Domain engineering and Application engineering were explained, and noted that variability management is an important challenge.

Then, these two fields of research were linked to each other, by the observation that MDE technology would be useful in building a software product line. It also stated that a step-wise, model refining transformation sequence, which is an MDE concept, can be regarded as a software platform, a PLE concept.

3

Case-study: Administrative, Web-based, Information Systems

3.1 Introduction

Throughout this report all the experiments are carried out on a transformation sequence case. This chapter describes this case, which also will be the context for most examples in this report. The case has proven to be a good way to validate the concepts that were developed. The transformation sequence is also called a software platform.

For the purpose of experimenting we developed a transformation sequence that generates web-based, administrative information systems with simple Create, Read, Update and Delete (CRUD) functionality and a persistency layer. The approach is comparable to the way web services are generated using the framework described by Xiaofeng et al. [YZZ+07], with the difference that the systems generated in this case are more client-side (ie. web-browser) oriented, and their software is more server-side oriented. However, their approach demonstrates that such an approach is feasible as it is improving development speed.

As far as motivation goes, the main motivation is very similar to the approach of Generative Programming by Czarnecki et al. [CE00], which is to *"improve reusability by providing parameterized components which can be instantiated for different choices of parameters"*. Also, like in Czarnecki et al., this case also involves *"modeling families of software systems by software entities such that, given a particular requirements specification, a highly customized and optimized instance of that family can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge"*.

These two statements say that the motivation is to configure a set of products reusing pre-fabricated generic components, and that the integration of these components should be automated. The model-driven approach from this thesis

has the exact same motivation. The configuration involves variability in components, models and modules, and this thesis describes how to handle this in a model-driven context. Our approach is less about parameterizing components, and more about specifying integration rules in a model transformation definition by matching on elements from the source model of a model transformation. That is, the model transformation definition is regarded as a starting point for managing variability, and configuring a generated member product.

3.2 Requirements

The (high-level) functional requirements of the transformation sequence and its generated family products are (amongst other, we list the most important here):

1. The transformation sequence shall generate an administrative application for a domain. The domain is described by an Ecore model, that is, a class diagram, and is the input for the transformation sequence.

2. For every class in the input model, a data overview shall be generated. There shall be buttons for adding new items, editing existing ones, and deleting items. The type of the attribute shall determine the editor for a field. There shall be support for different types of data overviews: a grid view, a tile view, and a tree view. One should be able to configure this on a per-class basis.

3. Data is to be persisted in a database. Two databases are supported: MySQL (relational database) and Google App Engine (object database, cloud-solution). It should be configured beforehand which one is to be integrated.

4. For each reference of a class A to a class B, the system shall have a drag and drop interface. This way, the user can connect B items to A items conveniently.

5. All data should support sorting (by clicking on the column headers), filtering, and searching.

6. Users should be able to login to and logout from the system.

Also, there are some quality requirements that are important:

1. The generated shall be expressed in the Google Web Toolkit language, and follow proven design patterns like Model-View-Controller (MVC).

2. The generated code shall be adaptable and extensible. If eventual systems need to be adapted or extended, it should be possible to do so without modifying the generated code, for instance by using inheritance.

3. The transformation specifications shall be extensible and adaptable. Specifically, it should be capable of generating systems according to a feature configuration, without having to change the transformation specification manually. Also, when extra features are added, the impact on other code should be low. Lastly, concerns should be separated as much as possible.

The quality requirements are the most important, but also the hardest to verify if they are achieved. We want to end up with a flexible software platform, that is highly configurable and flexible. This can be achieved when the code is adaptable and extensible.

For the eventual code, we use established object-oriented paradigms to achieve this (ie. MVC patterns), but it is also important that the ATL code has similar quality properties. If all steps of the transformation sequence have good qualities, there are more points where the systems can be adapted and extended, ideally in a non-invasive and low-impact way. This increases the flexibility of the transformation sequence as a whole, and thus increases the spectrum of potential information systems that can be generated.

3.3 Transformation sequence

The input of the transformation sequence is a class diagram. For example, the simple class diagram in Figure 3.1 will be transformed in a system shown in 3.2. The diagram conforms to the Ecore meta meta model, and contains domain entities. For each EClass C, the eventual system will contain a grid like overview of this class, with the attributes as columns, after which an instance of C can be created, read (viewed), updated (edited) and deleted.

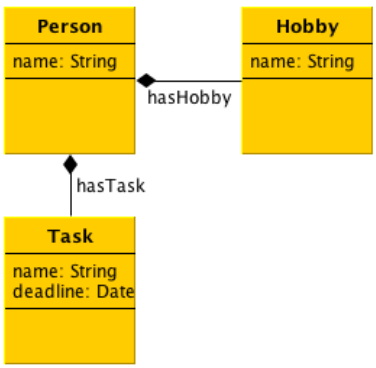

Figure 3.1: Sample input meta model, which conforms to the Ecore meta meta model.

The transformation sequence is shown in Figure 3.3. On the meta meta model level (M3), Ecore is shown. On the meta model level (M2), there are five different meta models:

- Input — This meta model conforms to Ecore and is the input meta model[1]. This model contains domain entities, like someone would model in an early stage of the development of an information system. Notice that this model is on M2, making the transformation diagonal. This is the reason this model is called 'input meta model' rather than the 'input model'.

[1]In this thesis, the terms domain model, input model, and input meta model are used interchangeably.

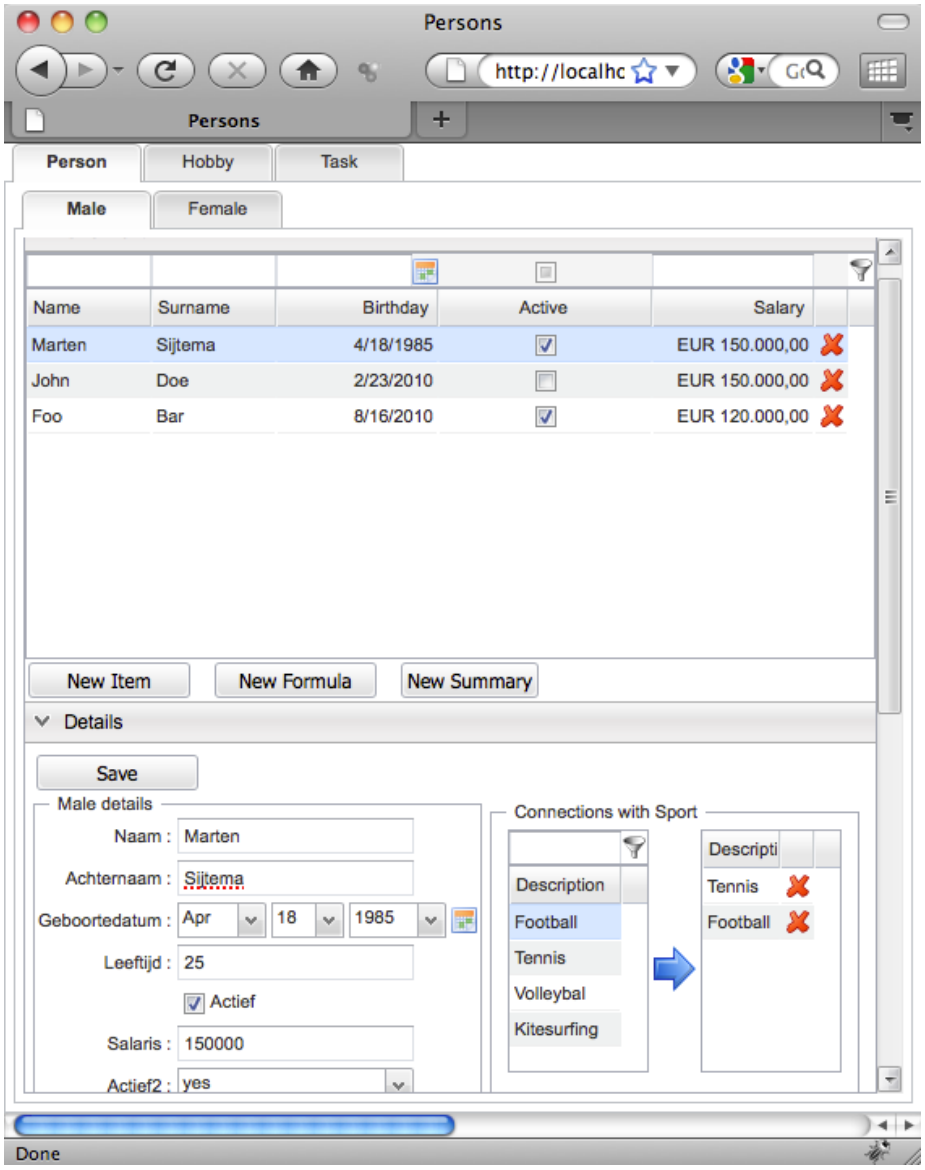

Figure 3.2: Screenshot of a system generated by the case.

- ATL — This is the meta model of the model-to-model transformation language ATL. An instance of meta model is an actual transformation. It transforms a model conforming to Ecore meta meta model into a model conforming to the CRUD system meta model.

- CRUD system — This is the meta model of a CRUD system implemented in an object oriented language, as shown in Figure 3.4. It typically contains Model, View and Controller artefacts (this means that the instances of this meta model will contain {Person, Task, Hobby}Model, {Person, Task, Hobby}View, and {Person, Task, Hobby}Controller in the case of Figure

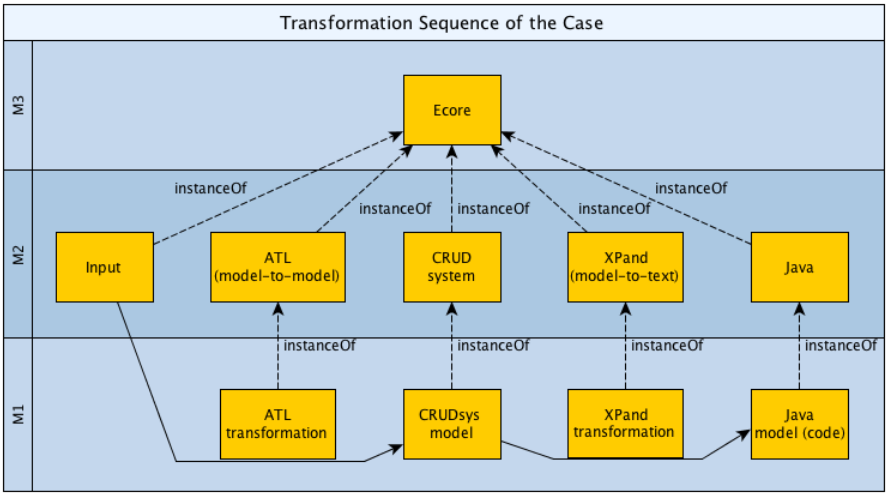

Figure 3.3: Transformation sequence of the case. Solid arrows shows the transformation order, dashed lines show instanceOf relationships.

3.1), as well as data access classes that can access data stores, to model persistency. This meta model can be enhanced with numerous features, and in the real-world it will be more elaborate than this prototype, but this example is kept simple for clarity.

- XPand — XPand is a model-to-text transformation language, that is based on the paradigm of templates. It is a very simple yet effective way to serialize models to text, in our case Java code. An instance of this meta model is a transformation template. It transforms instances of the CRUD system meta model to instances of the Java meta model.

- Java — The last meta model is a familiar one: Java. The system is actually implemented in the GoogleWebToolkit (GWT) framework, which shares its syntax with Java. It is a technology to provide a Java-like programming style (including all the mature Java tools like unit tests, debuggers, strong typing, etc.) for interactive web applications. The GWT compiler compiles this Java code into HTML/Javascript, which means there are no plugins required for running this application.

Let us look back at the CRUD system meta model from Figure 3.4. Observe that an instance of this will have a set of MVC classes instantiated. This means that a class from the input model is now transformed and refined into three parts: a Model, a View and a Controller. The model will contain a reference to a ModelEntry, this will keep the information of one item from the domain class. For instance, if the input model has a class 'Person', there now will be a PersonModel, PersonModelEntry, PersonModelView (or one/more of the subtypes) and PersonController. The ModelEntry has attributes, and references, which are copied from the normal references and attributes that exist in an Ecore model. In this example, a PersonModelEntry will eventually

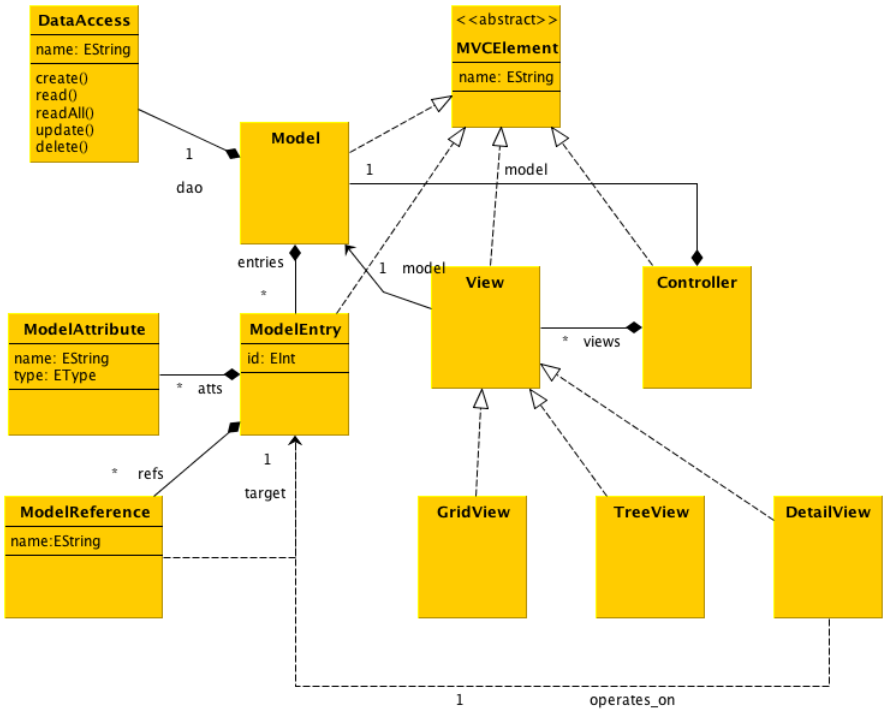

Figure 3.4: Simplified version of the CRUD system metamodel. It contains familiar patterns like Model-View-Controller and a data access class for accessing databases.

be a run-time Java object containing one Person. The PersonModel is there to group them together.

Finally, each Model class has a reference to a DataAccess class. This is a common pattern where the actual communication (reads and writes) with the data store is separated from the MVC pattern, making it is easy to swap database technologies. Also, because it is a web-application, the DataAccess classes will reside on the server, whilst the Model(Entry) instances reside on the client, in the browser.

The real CRUD system meta model is on the CD-ROM in the Appendix. It has much more details and contains other features.

3.3.1 Relation to Product Line Engineering (PLE)

The purpose of this transformation sequence is to be able to generate a family of information systems; CRUD based, data-oriented, web-based administration systems. It does not matter if one would like to manage car parts, books, people, music or appointments. We also would like to be able to generate a family of systems for one specific input meta model. In this last case, members could differ on things like database technology, color schemes, and the way data is presented.

Because of these characteristics and requirements of the transformation se-

quence, we state that it can be considered a product line. Therefore, PLE concepts and methodologies apply.

The (meta) models from the transformation sequence can be regarded as domain artefacts, resembling to the domain engineering phase of PLE. The application engineering is done by running the transformation sequence with a certain input meta model.

In the application engineering phase, there are multiple places where customizations can take place to yield a different member product. One could modify the transformation sequence (ie. the meta models or the transformations) to change aspects of a particular generated application, although this compromises the reusability of the transformation sequence. Alternatively, one could extend the generated code using normal object-oriented constructs like inheritance. The downside of this is that the abstraction level might be too low, and it neglects the MDE philosophy of bridging the gap between high and lower level of abstraction.

In this thesis the framework will be adapted such that the management of variability is easier, and does not only come down to modifying the generated code, or modifying the transformation sequence. Instead, the domain artefacts will be as generic and reusable as possible, where the concern of feature configuration is separated (ie. not integrated in) from the models in Figure 3.3.

3.4 Conclusions

We discussed the transformation sequence that acts as a software platform for generating web-based, administrative information systems. It consumes an Ecore input model, and is refined to an instance of a CRUD system meta model, which is then again refined into an instance of the Java meta model. It is important that the transformation sequence is extensible and adaptable, which also goes for the generated applications. This means that both the eventual Java code and the transformation definitions should have these qualities.

The goal is to get a software platform that is highly configurable, extensible, and well-maintainable. Ideally, extensions should be non-invasive and have a low impact on other parts of the system. The configuration mechanism should be separated as a concern.

The next chapter will zoom in on the variability that has to be managed in the case of an information system generating transformation sequence as discussed in this chapter.

4

Variability Management in Transformation Sequences

4.1 Introduction

Variability already was discussed in Chapter 2. However, this chapter will put the focus on the manifestation of variability in transformation sequences. In other words, it will classify the relevant types of variability that can occur in parts of the transformation sequence. As a context, the case of the previous chapter is used as example.

As an example variation point in the running example software platform (with a fixed input meta model), consider *database technology*. Say, one could choose a relational database or opt for an XML database. Different options like these are called **variants**, and the concern that groups these variants is called a **variation point**. We use the terms variants and features interchangeably, and consider them synonymous. Variability has to be managed, that is, there should be an effective way of modeling and implementing different variants in transformation sequences, and a particular set of selected variants should be automatically integrated in the generated system.

To model variability, we use the concept and notation of feature models [BPSP04]. Recall that in PLE, it is considered useful to treat variability as an orthogonal concern. That is, it is a phenomenon that can be separated from domain models, modularized in its own model. The variants (or features, as they are called) in the model can be mapped onto realization artifacts in the domain (meta) models.

The next section formalizes the requirements of the variability management concern in our case.

4.2 Requirements of our CRUD platform

We propose a formal way of explaining what our platform should be capable of, from the variability management perspective. It will show what is meant by variability in our example transformation sequence.

The software platform can be regarded as an injective function f, which produces a system s for a given input meta model x. For the rest of this section it holds that $x \in X$ where X is the set of all Ecore models, and $s \in S$, where S is the set of all information systems.

Thus, for the initial transformation sequence it holds that:

$$f : X \to S, \text{ where } f \text{ defined by: } f(x) = s \tag{4.1}$$

Because it is injective, when the input model is different, the resulting system will be different:

$$f(x') = s', \text{ where } s \neq s' \tag{4.2}$$

Of course, changing x could be regarded as a type of variability. This type, however, is not problematic, and the initial version of f is capable of dealing with such variability; x is just a parameter. The problem, however, is that for a fixed x, a (finite) *set* S' of systems should potentially be generated:

$$f(x) = S', \text{ where } S' \subset S \tag{4.3}$$

Since f is a function, and a function has at most one correspondence in the codomain for each input, more parameters are needed to be able to generate a set of systems. This yields f'. Here, the parameters are a feature model fm (with a particular feature set selected) and an annotation model am. The annotation model is explained later in this chapter, and is an additional source of feature configuration. It holds that fm is an element of the set of feature models FM: $fm \in FM$. Similarly it holds that $am \in AM$, AM being the set of all annotation models:

$$f' : X \times FM \times AM \to S, \text{ where } f' \text{ defined by: } f'(x, fm, am) = s \tag{4.4}$$

Now we can vary fm and/or am, but keep x fixed, allowing to generate different systems:

$$f' : X \times FM \times AM \to S, \text{ where } f' \text{ defined by: } f'(x, fm', am') = s', \text{ and } s \neq s' \tag{4.5}$$

The research question RQ2 (*"How can the MDE infrastructure be adopted to manage these types of variability, conserving good quality characteristics?"*) can be related to Equation 4.4, by replacing the words 'the MDE infrastructure by' f'.

4.3 Classification of variabilities in the transformation Sequence

With the feature model being orthogonal, the features from this model will correspond to realization artefacts (ie. classes, attributes or references) in meta

models, which can be instantiated into a model. In our case, this correspondence is made explicit by a transformation written in the model-to-model transformation language, ATL. In other words, all the possible variants from the feature model will correspond to a set of realization artefacts which reside in meta models, and ATL provides the mapping.

The mapping is not only a trace, but ATL will instantiate and integrate the realization artefacts once a mapping is there, which allows us to do the configuration of the family members of the product line with the feature model as driver. Following that, the transformation sequence can generate an application with the selected features integrated.

However, sometimes the feature model alone is not enough as a configuration driver, sometimes also information from the source model (of a model-to-model transformation step) is needed. To illustrate, we consider two types of variability that may occur in the example case product line: the variants either depend on configuration information from the source model of a model-to-model transformation step, or they don't. These two cases are respectively called *source model dependent* and *source model independent* variability, and are explained by example in the next two subsections. Ideally, the transformation language ATL should be able to handle both cases efficiently.

Then, we can achieve what is formally described in Equation 4.4.

4.3.1 Source model independent variability

In source model independent variability there is only a mapping (for the concern of variability configuration) between feature realization artefacts from a meta model and the feature model. That is, there is no mapping needed between the feature model and the input model. As an example consider the variation point database technology as pointed out in the introduction of this chapter. It has two alternative implementations: a relational database and an XML-database. Now, no matter what the source model looks like, only the feature model's selection (ie. relational *xor* XML is selected) determines what realization artefacts from the target meta model are instantiated in the resulting generated model of the transformation step.

Typically, variants that are technology related –and thus have system-wide implications– fall into this category.

Comparing it to the formal definitions from Section 4.2, this means that if an ecore model x, a feature model fm and an *empty* annotation am are put into f' from Equation 4.4.

Only an input model and a feature model are needed to produce a system using the platform.

4.3.2 Source model dependent variability

Variability that falls into this category *does* depend on the source meta model. For example, in the feature model we support two alternative ways of presenting the data to the user, a Grid view and a Tree view. So, if the source model has a class 'Person', which has a one-to-many reference to a class 'Task', it means that the eventual system will store persons and tasks, and these lists of person and task data will be shown in either a grid or a tree view. This means that the target meta model will have a realization artefact for a GridView and one for a

TreeView. It could very well be that we want to instantiate a GridView for the Person class, but a TreeView for the Task class. So, without information from the source model, the transformation cannot be completed, which differs from the *in*dependent case.

Source model dependent variability is also modeled in the feature model, but cannot be 'selected' or 'deselected', as is the case with independent variability. Instead, the source model should be annotated with the feature, in order to provide the information needed by ATL in the transformation step.

As Section 4.5 will explain, we use a separate annotation model for annotating a model. The approach of [VBVM09] has shown that this is a useful way of parameterizing a model, without cluttering it with the annotation.

4.4 Feature diagrams for the case

This section provides some example feature models and explain how they manifest in the transformation sequence. Figure 4.1 and 4.2 show (a selection of) independent and dependent variabilities respectively. Note that *these could be mixed in one model*, in fact, the complete feature model of our transformation sequence mixes these types into one feature model. The main responsibility of the feature model is to constraint the selection space. As mentioned, there is no such thing as 'being selected' for source model dependent features. However, it does make sense to model it to explicitly state the selection constraint. Section 7.3 shows why.

Consider the variants shown in the independent variability model of Figure 4.1. As stated before, these variants are not (from a configuration point of view) related to entities in the source meta model. Often, such features are technology related or reason about overall (ie. not per-EClass of per-EAttribute) behaviour/structure, meaning that every feature can be selected or deselected, regardless of the source meta model's structure.

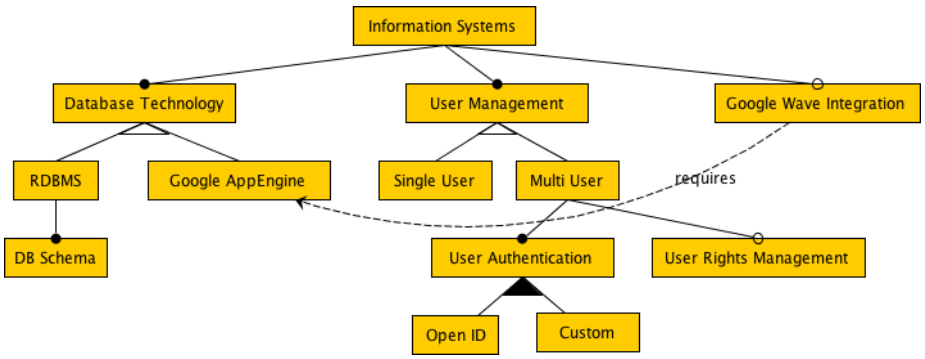

Figure 4.1: Feature model with only source model independent variability.

This is not the case for the dependent variabilities. Figure 4.2 shows dependent variability. These features could be mixed into the model from Figure 4.1, but are isolated here for clarity.

The Person class from the source meta model will be connected to a GridView, whilst the Task will be connected to a TreeView. Note that this model

does *not* show *how* the Grid and Tree views are realized, because the realization artefacts of Grid and Tree will reside as meta entities (classes, references, attributes, etc.) in the target meta model. It is the responsibility of the ATL model-to-model transformation to instantiate these views and connect them to (copies of) the appropriate entities from the source meta model. This example shows that such features cannot be selected or deselected, as was the case with the independent variabilities. It could very well be that a Grid view is 'selected' for the Person class, but 'deselected' for the Task class. Consequently, if the source model is replaced by a different model, a new mapping has to be made, in order to be able to run the transformation.

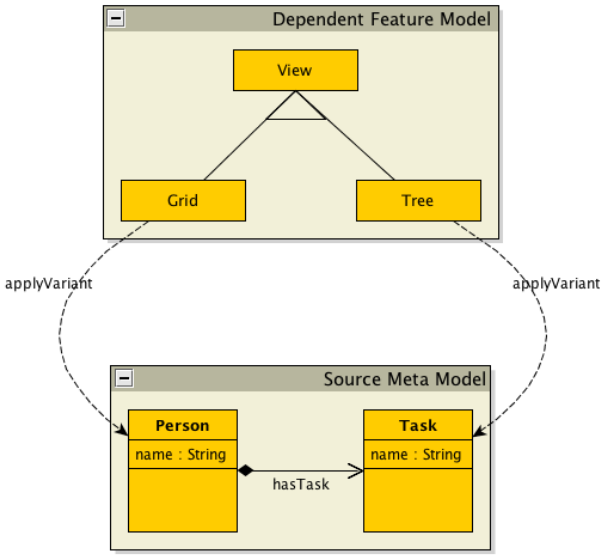

Figure 4.2: Feature model with source meta model *dependent* variabilities. In the source meta model drawn below, the Person data entity will be using a Grid view, whilst the Task entity will be using a Tree view.

The fact that a feature is dependent or independent cannot be concluded from the feature model alone. It will be inferred from the annotation model; if an entry for a certain feature exists there, it is regarded as a dependent feature. Similarly, the non-existence implies that the feature is independent. This leaves the feature model oblivious from these concepts, allowing the use of any feature model representation.

4.5 Annotation models

The approach of annotating an input model will be with a separate annotation model. The approach of [VBVM09] has shown that this is a useful way of parameterizing a model, without cluttering it with the annotation. An annotation model a is a model that annotates a different model x. It matches a (set of) elements from x, and puts one or more annotations on this. One could for instance

express in a modeling language which creates annotation models something as follows ($x \in instanceOf(Ecore)$): 'annotate all EClasses with name "A", "B" or "C" with annotations "feature 1", "feature 2" and "feature 3"'.

This way, the annotations are separated from x. Preferably, a modeling language with a declarative syntax is able to create such models. Our approach will be to have a syntax which can be compared to Cascading Style Sheets (CSS) [CSS], the popular language for decorating HTML elements (ie. < *table* >, < *div* >, < *ul* >, etc.) with style information (ie. *font-size*, *background*, *border-color*, etc.).

Figure **??** shows an example textual representation.

```
1 EClass [name=''Person''] {
2         Grid, otherFeature1
3 }
4
5 EClass [name=''Task''] {
6         Tree
7 }
```

Listing 4.1: Example annotation model.

4.6 Conclusions

To conclude this chapter, two types of variability are identified, and these should be handled by the model-to-model transformation ATL. It also proposed to model the variability separately in a feature model. For the dependent variability case, extra information is needed in the form of annotations on the input model. Annotations reside in a separate annotation model, to prevent the input meta model from being cluttered. The next chapter will explain how this is achieved, by first showing that the current version of ATL is not well-suited to handle the configuration requirements. Therefore, the next chapter introduces a new ATL concept, called *variability rules*, which solves the issue of managing these two kinds of mappings.

5 Extending ATL with Variability Rules

5.1 Introduction

This chapter introduces first-class variability management means for the model transformation language ATL [JK06], using a separate feature model for configuration. First, the management of independent variability is shown, since it is the simpler case. Later, in Section 5.5, the implementation is extended to handle source model dependent variability.

As a starting point, the basic modular construct of rule-based model transformation languages is used: a rule. In ATL, there are five types of rules: normal rules, abstract rules, rules that inherit from other rules (ie. sub rules), lazy rules and called rules. This chapter introduces **variability rules**. Ideally, variability rules should have the same quality properties as normal rules. Parts of this chapter have been published for the MtATL2010 workshop on ATL transformations [Sij10].

5.2 Concept and Motivation

Variability rules are best used in the context of a transformation sequence which successively refines models, and where blueprints of feature realization artefacts reside in the meta models, as is the case in our running example transformation sequence. Thus, meta models define the complete product space by defining feature realizations for every feature. The variability rules instantiate and integrate a particular set of feature realizations from the meta models into a final product, according to the feature model's selection.

The reason for using a transformation rule as key construct for managing variability is the observation that features from a feature model have to be associated with, or mapped to, feature realization artefacts in order to enable automatic product derivation. Specifically, the feature model should be the driver for configuring/assembling a product family member. Rules are purpose-

built for defining mappings. Furthermore, since rules have mature mechanisms for matching, querying and creating (instances of) meta model elements, and thus (instances of) feature realization elements, they are well-suited for integrating feature realizations with the rest of the system. Furthermore, rules are modular constructs with a declarative nature, which makes them easy to use and allow elegant implementations as opposed to more imperative solutions.

5.3 Problems in the current ATL

Recall that as a requirement we would manage the variability in the model transformation definition, while conserving quality properties like extensibility and adaptability (see research question RQ2). Also, we stated that new features could be added in a non-invasive way, with low impact on other code (see the quality requirements in Section 3.2). Lastly, declarative code should be as used as much as possible, and imperative code should be avoided. On these points there are some problems in the current situation, as we explain here; it explains the current shortcomings of ATL when it comes to managing variability.

Model transformations are a central concept in MDE, and by using a transformation sequence as a software product line generator, model transformations should be capable of dealing with variabilities.

Recall the running example case where input meta models conforming to Ecore are transformed into a model of a web-based, data-centric information systems with basic create/read/update/delete (CRUD) operation support, which is in turn transformed into code. The model-to-model step is shown in Figure 5.1. The meta model *outMM* is an excerpt of the real case meta model, and only shows a *Model* class (note: a Model from the Model-View-Controller pattern), as well as a *DatabaseTechnology* element.

Every derived instance model of *outMM* will contain a *Model* element *for each* instance of an *EClass* element from the source meta model *inM*. This *Model* element will have the same name as the *EClass*'s name attribute, concatenated with the string 'Model'. In this case, there is only one instance of *EClass* in the source meta model, *Person*, resulting in the instantiation of a *Model* element with the name attribute set to 'PersonModel'. Note that a different input meta model is likely to have multiple instances of *EClass*, and will therefore transform in a model with multiple instances of the *Model* meta class. In the real case, an MVC pattern is generated for each *EClass*.

The variability in this case (apart from varying the input model) is the type of database technology that a resulting application uses. There are two variants, as shown in the feature model *FM*: a relational database denoted by *RDBMS*, and an object database *GoogleAppEngine*. The meta model *outMM* shows that each *Model* element has a *DatabaseTechnology* element aggregated.

In the feature model *FM* of this example, the *RDBMS* variant is selected, so the resulting model *outM* has an instance of an *RDBMS* realization artefact for each *EClass* source element.

The example is used to show that normal ATL rules are ineffective in managing variability in the case, as the next attempts will show.

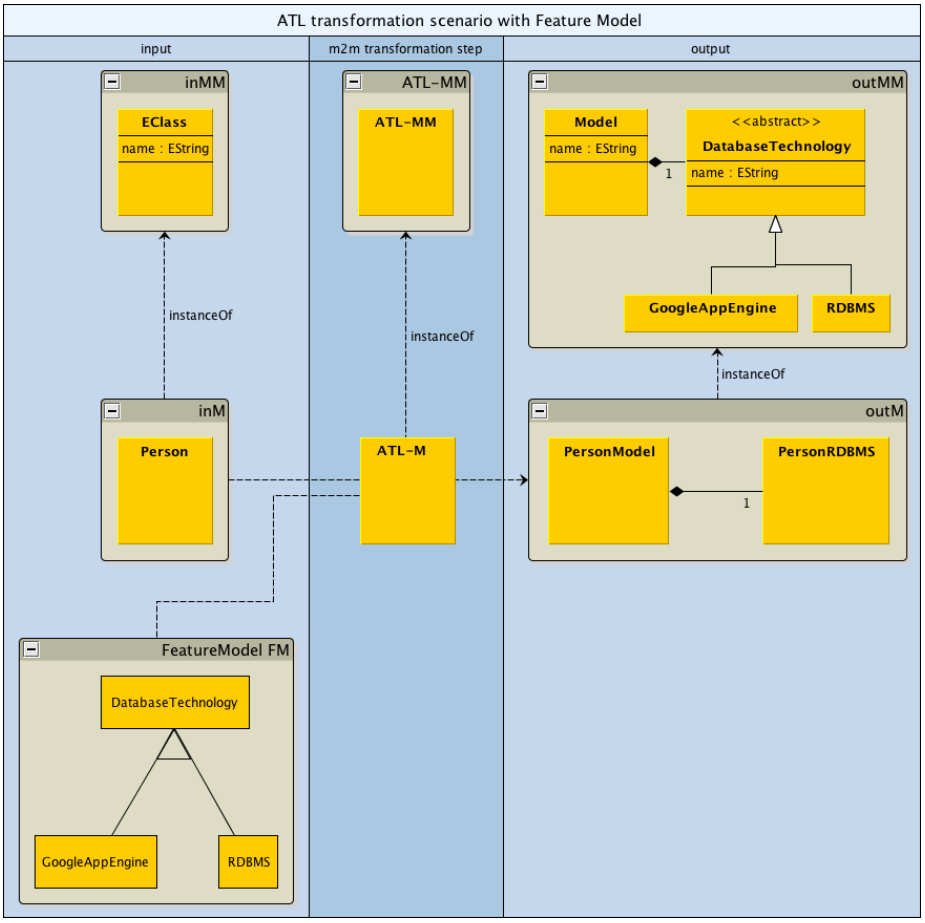

Figure 5.1: Typical model-to-model transformation scenario, using ATL and a feature model FM.

5.3.1 Solution 1 — Using Rule Inheritance

In ATL, rule inheritance could be used for guiding the feature assembly process in the above example, as shown (partially) in Listing 5.1.

```
1  abstract rule EClass2Model {
2      from
3          a:    inMM ! EClass
4      to
5          model : outMM ! Model,
6          db : outMM ! DatabaseTechnology
7  }
8
9  rule EClass2Model_RDBMS extends EClass2Model {
10     from
11         a:    inMM ! EClass (true)
12     to
13         model : outMM ! Model,
14         db : outMM ! RDBMS
15 }
16
```

```
17 rule EClass2Model_GoogleAppEngine extends EClass2Model {
18     from
19         a: inMM ! EClass (false)
20     to
21         model : outMM ! Model,
22         db : outMM ! GoogleAppEngine
23 }
```

Listing 5.1: Using rule inheritance in normal ATL to deal with variability management is insufficient.

Note that the *from*-clause in the rule *EClass2Model* matches on *inMM!EClass*, stating that the instantiation of the feature realization artefact should be done *for each EClass* instance. We use rule inheritance to instantiate a specific variant.

There is a sub rule for each feature, both specializing the rule *EClass2Model* with a specific database technology. According to the selected feature, here simply switched by *true* if selected, and *false* otherwise, the correct sub rule is called, and the super rule's content is inherited.

This situation shows that feature realizations often have to be integrated into the common base, *according to some rationale*, here for each *EClass*. Therefore we used rule inheritance, which seems a necessary strategy, but the problem with this is that ATL does not support matching a single source model element by more than one rule. So, what if there are more features, apart from database technology, that also have to be instantiated *for each EClass*? Or what if both features are allowed, because they are, for instance, both optional (ie. both *from*-clauses are set to *true*)? Then, multiple rules should be called for a single source element, which is not supported.

Note that also, the true/false must be switched manually here, or helpers have to be implemented that navigate the feature model and check if a particular feature is selected. So, the reusability of this model transformation is low, and changing a feature selection means changing the transformation definition. Ideally, we would have a transformation specification that is not affected when features are changed.

5.3.2 Solution 2 — Adding a feature model

Solution 1 could be extended, to also make use of a feature model FM as input. Then, one could create a boolean expression in each *from*-clause returning a boolean that is true if the feature is selected in FM and false otherwise, as shown in Listing 5.2, line 13 and line 24. Although this prevents from having the selection mechanism in the transformation definition, the problem of matching multiple rules for a single element remains. So we gained a bit here, but there is still a fundamental problem.

```
1 abstract rule EClass2Model {
2     from
3         a: inMM ! EClass
4     to
5         model : outMM ! Model,
6         db : outMM ! DatabaseTechnology
7 }
8
9 rule EClass2Model_RDBMS extends EClass2Model {
```

```
10      from
11          -- isSelected queries the Feature model,
12          -- to see if feature is selected
13          a:  inMM ! EClass (thisModule.isSelected('RDBMS'))
14      to
15          model : outMM ! Model,
16          db : outMM ! RDBMS
17  }
18
19  rule EClass2Model_GoogleAppEngine extends EClass2Model {
20      from
21          -- isSelected queries the Feature model,
22          -- to see if feature is selected
23          a:  inMM ! EClass
24          (thisModule.isSelected('GoogleAppEngine'))
25      to
26          model : outMM ! Model,
27          db : outMM ! GoogleAppEngine
28  }
```

Listing 5.2: Using rule inheritance in normal ATL to deal with variability management is insufficient even when a Feature model is added.

This solution would be usable in model transformation languages that *do* support matching multiple rules for a single element. However, as mentioned, the ATL model transformation language is popular, so there is motivation to have variability management means in ATL as well.

We just stated that it is usable, but a good variability management solution would ideally abstract the user from checking a feature model manually. We prefer a solution where the variable parts are specified declaratively in a modular construct like a rule, yielding a stable transformation specification. This solution would come close however, with the main problem not being able to match multiple rules for a single element.

5.3.3 Solution 3 — Matching on the Feature Model Elements

As an alternative, a rule which matches on elements from the feature model FM could be used, instead of matching on the input meta model (Listing 5.3). Such a solution usually avoids having different rules with the same *from*-clause. But doing something 'for each EClass' is now not possible, which was specifically intended.

```
1  rule selectRDBMS {
2          from
3                  a : FM ! RDBMS (a.isSelected)
4          to
5                ...
6  }
```

Listing 5.3: Selecting on elements from the feature model breaks the intent of the transformation developer.

This point is important. Writing transformations in ATL consists for a large part of matching source elements in the *from*-clause. Writing a rule actually starts from there, thus a transformation developer would have reasons to match

on a set of elements, and group what is to be done for this matched set in a rule. The fact that the developer has to manage variability, should not make him or her have to do tricks (ie. writing OCL-expressions that iterate over sets of source elements) to achieve the same result! It would be too intrusive, and would complicate the model transformation definition.

One could use imperative code, which is not recommended. As mentioned, we prefer declarative, modular constructs.

Thus, there is not a good way to let features guide a model transformation, at least not without resorting to imperative ATL code. We provide a more elegant, easy to use solution, by extending ATL with the concept of variability rules, as the next section will explain.

5.4 Managing Source Model Independent Variability

This section describes the concept of variability rules. The syntax and semantics are not much different from normal ATL rules. The version discussed here provides means to manage source model independent variability, the simpler case. Later, in Section 5.5 the case of source model dependent variability is discussed.

A **variability rule** looks like a normal rule, preceded by with the *variability* keyword:

```
1  variability rule RDBMS configures EClass2Model {
2      from
3          -- same as EClass2Model's from clause
4          a : inMM ! EClass
5      to
6          db : outMM ! RDBMS (
7              name <- a.name+'RDBMS'
8          )
9  }
10
11 variability rule GoogleAppEngine configures EClass2Model {
12     from
13         -- same as EClass2Model's from clause
14         a : inMM ! EClass
15     to
16       db : outMM ! GoogleAppEngine (
17         name <- a.name+'GoogleAppEngine'
18       )
19 }
```

Listing 5.4: Extended concrete syntax of ATL with variability rules.

Observing the example from Listing 5.4, the syntax is a bit different than normal rules. In the first line, 'RDBMS' refers to the feature from the feature model. Secondly, 'EClass2Model' refers to a normal rule. The variability rule is said to 'configure' a normal rule. Finally, the *from*-clause is identical to the one in ECLass2Model.

5.4.1 Semantics

The semantics of variabiilty rules is as follows:

- A variability rule 'configures' a normal rule, specializing its implementation, just like rule inheritance,
- a variability rule is executed for each match, but if and only if the corresponding feature is selected in the feature model,
- multiple variability rules can have the same *from*-clause,
- multiple variability rules can configure the same normal rule.

Thus, specific feature realization artefacts can be instantiated from the target meta model. Also, multiple rules that match on a single source model element can be called. This solution is very comparable to a conventional declarative mapping in a normal rule, the implicit execution order property still holds. Furthermore, the concern of selecting features is now cleanly separated. It is the feature model editor's responsibility to facilitate/constraint the selection. Variability rules act as a template; a modular piece of code that is not part of an (explicitly defined) imperative process, used for instantiating a particular feature.

Consequently, a developer can write one or more variability rules for each feature, without having to worry about concerns like selection constraints or execution order. The developer can just declaratively define what should be instantiated if a certain feature is selected.

The next section shows how variability rules are implemented.

5.4.2 Implementing Variability Rules using a Higher-order Transformation

The variability rules are implemented by a higher-order transformation (HOT), as shown in Figure 5.3. The HOT is created as follows:

1. The ATL meta model was extended, yielding the ATL' meta model. The extension is not large; one EClass was added, as shown in Figure 5.2. The *configures*-clause points at the $MatchedRule$ that is extended, just like a sub-rule would with the *extends*-clause, where it points at a $MatchedRule$. The name of the variability rule is stored in the attribute $variantName$ and denotes the name of the feature in the feature model.
2. The concrete syntax of ATL was extended with variability rules, a corresponding editor is created. We used the tool TCS (Textual Concrete Syntax) for the syntax extension, which generates editors for concrete syntaxes. The syntax is based on the extended ATL meta model, ATL'. Section 5.4.3 explains briefly how this is done.
3. A HOT was developed in ATL which transforms an ATL' model into an ATL model. The workings of this HOT are explained below.

The HOT will be a used as preprocessing step in a transformation sequence. The resulting ATL model is subsequently executed, to yield the aimed result.

The compilation step of the HOT works as follows. First all the *to*-clauses of variability rules whose feature is selected are gathered (ie. union), and these are grouped by the rule they configure. Then, these consolidated *to*-clauses are

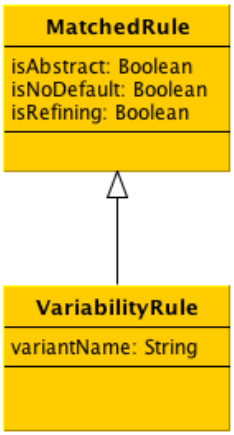

Figure 5.2: The point where the ATL meta model was extended to yield the ATL' meta model.

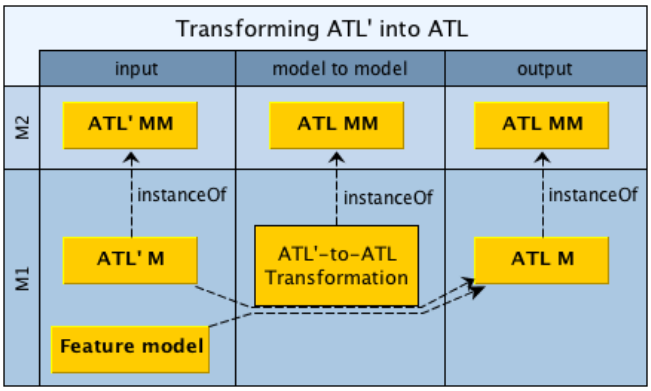

Figure 5.3: A feature model together with a variability rule enriched ATL file (ATL' M) are transformed into a native ATL file with only the selected features transformed into normal ATL rules.

added to the normal rule that is configured. This results in a set of normal rules whose *to*-clauses only contain the *to*-clauses from the relevant variability rules, effectively resulting in a transformation that transforms a source model into a target model containing the selected features. In other words, the result is semantically equivalent to executing multiple rules that match on the same source model element. In the current version, no error-handling is done when merging *to*-clauses, like variable name clashes. This is left as future work, and should be added when industrializing the concept.

Figure 5.4 shows this effect of the HOT schematically.

Figure 5.5 shows how the HOT is integrated in the complete transformation step. Observe that it acts as a preprocessor.

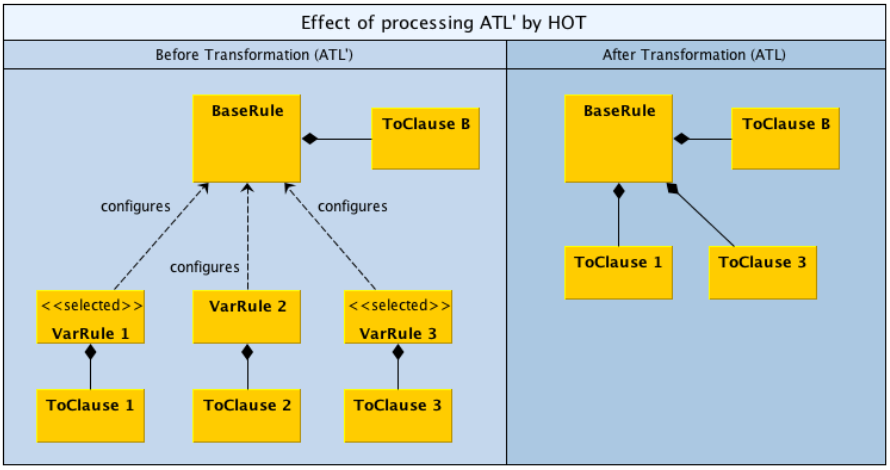

Figure 5.4: Effect of running the HOT with independent variability rules. The to clauses of the selected variability rules are merged into the base rule.

Note that since there is no de-facto standard for feature (meta) models, we used our own. If one would want to use its own, a custom version of the (simple) ATL helper function $isSelected(f : Feature) : Boolean$ (residing in the HOT) should be included in the higher-order transformation.

5.4.3 Extending the Concrete Syntax using TCS for Independent Variability

The ATL meta model was extended into ATL'. Since ATL and ATL' conform to the Ecore meta meta model, extension is done by the inheritance relationship. The ATL meta model was extended on the EClass *MatchedRule*. For this, a *VariabilityRule* EClass was built, as shown in Figure 5.2.

Next, the original ATL.tcs template file was downloaded and modified. The TCS tool now generated an ATL' editor, parser and injector, which could be run as an Eclipse plugin. Listing 5.5 shows what was added to the ATL.tcs file, yielding the new concrete syntax. Note that the *superRule* reference is used to be able to inherit variables from configured base rules, meaning that variability rules have the same characteristics as sub rule (from rule inheritance).

```
1  template VariabilityRule context addToContext
2          : "variability" "rule" variantName
3              "configures"
4                  (isDefined(superRule) ?
5                   superRule{refersTo=name,
6                         importContext}) "{" [
7  inPattern
8  (isDefined(variables) ?
9      "using" "{" [
10         variables
11     ] "}"
12 )
13 (isDefined(outPattern) ? outPattern)
14 (isDefined(actionBlock) ? actionBlock)
```

```
15                ] "}"
16              ;
```

Listing 5.5: The TCS template file of ATL' where the extended syntax is specified

5.5 Managing Source Model Dependent Variability

At this stage, the implementation of variability rules cannot handle source model dependent variability, because there are no means to define a mapping between a feature and a particular source meta model element.

Therefore the syntax and semantics of variability rules are extended further, as shown here. The approach uses annotations to define which features should be applied to a given class, reference, or attribute (the annotation method is shown in Section 5.5.4, where a separate annotation model is introduced). For example, one could annotate the class Person with << *Grid* >>, to let it have a Grid view. Thus, feature selection is done in the source meta model, which differs from the independent variability case, where the feature selection is done

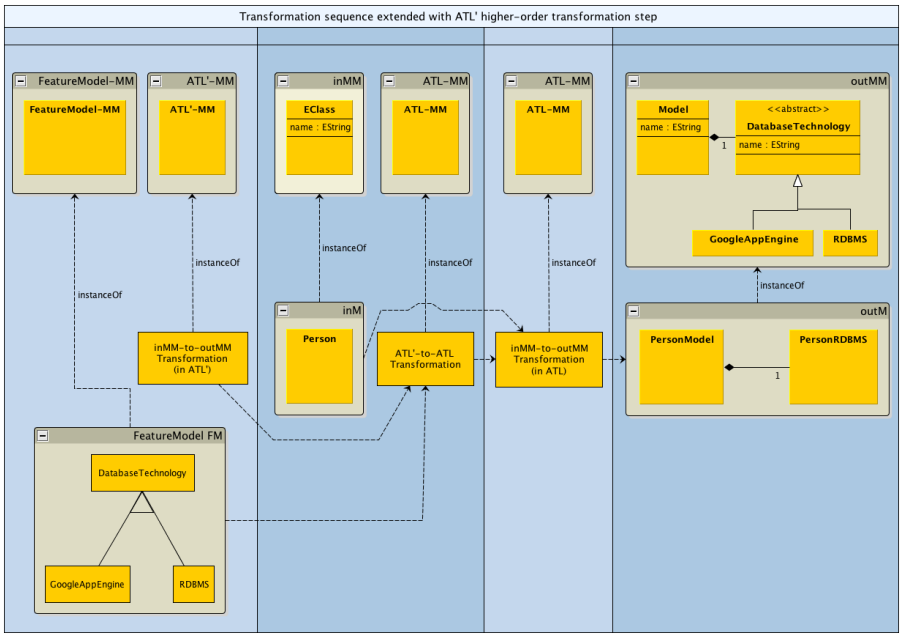

Figure 5.5: The complete transformation sequence, extended with higher-order transformation. This transformation first transforms an ATL' and Feature model into a normal ATL model. This ATL model consists of the original rules from the ATL' model, where the effects from the variability rules whose features are selected in the feature model are copied and consolidated into the normal rules they configure.

by setting the boolean variable *isSelected* in the feature model.

5.5.1 Extending Syntax of Variability Rules Further

The variability rules shown in Listing 5.6 (line 15 and 28) show a small syntax change; next to referring to a rule that is configured, also a target meta element is specified. Figure 5.6 shows the target meta model of the transformation. As shown in the rule $EClass2Model$, a $Controller$ element is created in the *to*-clause, identified by the variable *c*. The variability rules point to this variable by the dot-notation in the *configures*-clause, and create a $GridView$ respectively $TreeView$ element.

This is the only syntax change, compared to the normal variability rules. In fact, the dot-notation classifies a variability rule as dependent variability rule, whereas omitting it assumes that the rule will manage independent variability.

Both View elements have to be connected to the eventual $Controller$ *c*, *if-and-only-if the element matched in the from-clause is annotated with 'Grid' respectively 'Tree'*. Note that *Grid* and *Tree* are the name of the variability rule, and these are string-matched on the feature model, thus have to exist in the feature model to be effective.

In the target meta model, the relation between $Controller$ and $View$ is called *views*. Therefore, the variable names of these elements in the *to*-clause of the variability rules are also called *views*. These elements can have sub-elements, so the *to*-clause can be more complex. As long as the *views* variable exists and is the root of the elements of the *to*-clause, this works.

If the feature model allows it (ie. Grid and Tree are not alternatives, but both optional for instance), one could also connect both views to the controller. Then, the EClass *a*, in the *from*-clause of the base rule, has to be annotated with both $Grid$ and $Tree$.

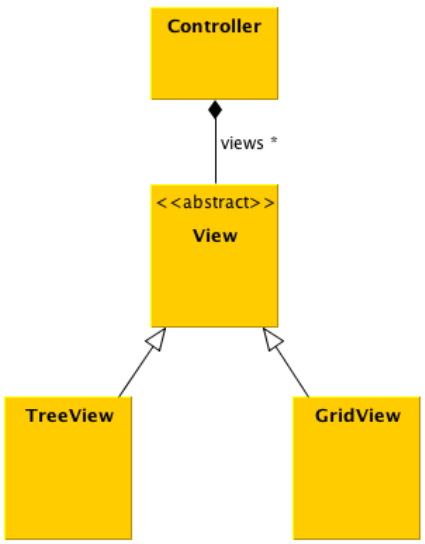

Figure 5.6: Excerpt of target meta model with a controller which contains views by the *views* reference

The target meta element is the point where the elements from the *to*-clause of the variability rules will be bound. So, after running the HOT, the base rule *EClass2Model* will create a *Controller* instance, which has a binding for the appropriate feature. This binding is dynamic, ie. at transformation run-time it is determined if the EClass *a* is annotated with *Grid* or *Tree*. The next section explains how the binding works, by showing how the variability rules are compiled into normal ATL constructs.

```
1  --Base rule
2  rule EClass2Model {
3          from
4              a: inMM ! EClass
5          to
6             c: outMM ! Controller (
7                  ...
8                  -- After running the HOT, the views are
9                  -- bound here, since the variability
10                 -- rules specified this
11            )
12
13 }
14
15 variability rule Grid configures
16               EClass2Model.c {
17     from
18         a : inMM ! EClass
19     to
20         --The 'views' variable will be woven
21         --in the base rule's Controller
22         --after running the HOT
23         views: outMM ! GridView (
24            name <- a.name+'TreeView'
25         )
26 }
27
28 variability rule Tree configures
29               EClass2Model.c {
30     from
31         a : inMM ! EClass
32     to
33         --The 'views' variable will be woven
34         --in the base rule's Controller
35         --after running the HOT
36         views: outMM ! TreeView (
37          name <- a.name+'TreeView'
38         )
39 }
```

Listing 5.6: Extended concrete syntax of ATL with variability rules again to handle dependent variability.

5.5.2 Resulting Code after Running the HOT

After the HOT has run, the resulting ATL model will look as shown in Listing 1.4. Every dependent variability rule will be compiled into a lazy rule. The lazy rule is called by the rule that is configured. The point from which it is called is, in this case, the Controller with variable c, as the *configures*-clause stated. The

lazy rule call is part of an if-then-else expression, where the rule is only called if the source model element isAnnotatedWith a certain feature.

This if-then-else statement is dynamic (ie. not hardcoded) from the perspective of the compiled ATL model, because it is the nature of the mapping. Hardcoding the binding is not possible, since it can differ per EClass in this case. Thus, the binding needs to be postponed to run-time of the ATL model that is resulted from the HOT. The reason is that now the HOT (as shown in Figure 2) does not require the input meta model as an input; just the feature model and the ATL model. This makes the HOT more modular, and reusable. Also, the HOT does not have to be run again, after the annotation model changes.

```
1  --Base rule, after the HOT has run
2  rule EClass2Model {
3          from
4              a: inMM ! EClass
5          to
6             c: outMM ! Controller (
7                 ...
8           views <-if
9                   thisModule.isAnnotatedWith(a, 'GridView')
10                  then
11                       thisModule.Grid(a)
12                  else
13                      Sequence {}
14                  endif,
15          views <-if
16                  thisModule.isAnnotatedWith(a, 'TreeView')
17                  then
18                       thisModule.Tree(a)
19                  else
20                      Sequence {}
21                  endif,
22             )
23
24 }
25
26 -- The variability rules are
27 -- compiled into lazy rules
28 lazy rule Grid {
29     from
30         a : inMM ! EClass
31     to
32         views: outMM ! GridView (
33            name <- a.name+'TreeView'
34         )
35 }
36
37 lazy rule Tree {
38     from
39         a : inMM ! EClass
40     to
41         views: outMM ! TreeView (
42          name <- a.name+'TreeView'
43         )
44 }
```

Listing 5.7: The HOT compiles the variability rules to lazy rules and the bindings are done in the specified meta element in the base rule.

The reader might have noticed that our approach allows for illegal feature

selections, if the annotations in the input meta model are wrongly put. To see why, assume that the class Person has two annotations: << *Grid* >> and << *Tree* >>. But the feature model states that these are alternatives, so only one is allowed. The check can no longer be the responsibility of the feature model editor, as was the case with source model independent variability. In fact, the semantics of what is allowed and what not can be interpreted in various ways: (i) a Grid and a Tree are alternatives per matching element (in this case inMM!EClass), or (ii) model-wide, ie. in the whole model there can be either Grid xor Tree views. Situation (i) is the most useful in the example case, as the examples show; there is much use of integrating features on a per class basis. The validity check, however, is left as future research.

5.5.3 The new ATL' Meta Model and TCS template

To incorporate dependent variability rules, the ATL' meta model and TCS template were extended further. Figure 5.7 and Listing 5.8 show these extensions.

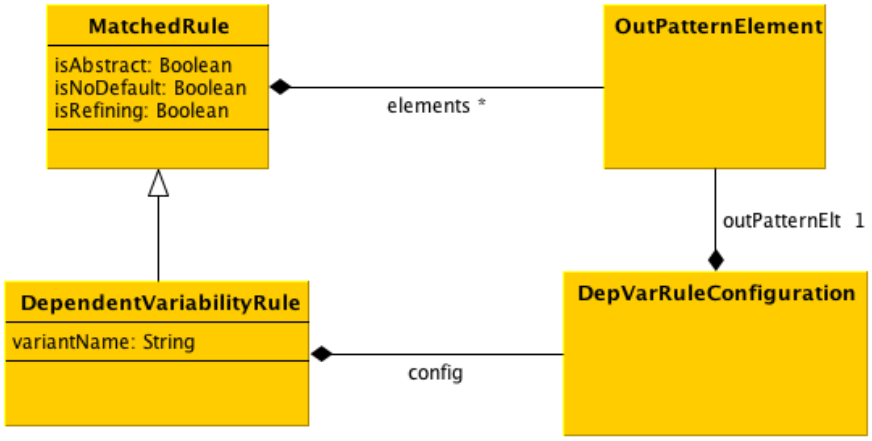

Figure 5.7: The point where the ATL meta model was extended to yield the ATL' meta model. This version incorporates dependent variability rules.

```
1  template DependentVariabilityRule context addToContext
2     : "variability" "rule" variantName "configures" (
3          isDefined(superRule) ? superRule{
4              refersTo=name, importContext
5           }
6       ) "." (isDefined(configuration) ?
7          configuration{separator=","})    "{" [
8          inPattern
9          (isDefined(variables) ?
10                "using" "{" [
11                      variables
12                ] "}"
13         )
14         (isDefined(outPattern) ? outPattern)
15         (isDefined(actionBlock) ? actionBlock)
16          ] "}"
17         ;
```

```
18
19 template DepVarRuleConfiguration
20                 context addToContext
21     : [ (isDefined(outPatternElement) ?
22      outPatternElement{refersTo=varName, importContext}
23       ) ]
24 ;
```

Listing 5.8: The TCS template file of ATL' where the extended syntax is specified now has support for dependent variability rules

The *DepVarRuleConfiguration* element from Figure 5.7 is a necessity for creating the concrete syntax in TCS. Specifically, it is necessary to define the separate *template*-construct in line 19 of Listing 5.8.

5.5.4 Annotation of the models — A separate annotation model

The approach of annotating an input model will be with a separate annotation model. The approach of [VBVM09] has shown that it is a useful way of parameterizing a model, without cluttering it with the annotation.

It is convenient to define a simple concrete syntax for creating the annotations, as the example Listing 5.9 shows. Here, a model element of the type *EClass*, named *Person* is annotated with the features *Grid*, and *otherFeature1*. A different EClass, with name attribute *Task*, is annotated with *Tree*. Furthermore, a model element of type *EReference* named *hasTask* is annotated with feature *otherFeature2*. The syntax is inspired by the Cascading Style Sheet (CSS) language used to decorate web-pages with style information like font-size, borders, background-color etc., in a non-invasive, declarative way.

```
1  EClass [name=''Person''] {
2          Grid, otherFeature1
3  }
4
5  EClass [name=''Task''] {
6          Tree
7  }
8
9  EReference [name=``views''] {
10         otherFeature2
11 }
```

Listing 5.9: Concrete and simple syntax of defining annotations.

The concrete syntax is developed by the tool called XText [oAW], which generated a parser and an editor. The grammar is shown in Listing 5.10, and the meta model is shown in Figure 5.8. This meta model is generated by XText. This is the difference between XText and TCS, XText infers the meta model from the grammar, while with TCS, one starts with making a meta model.

Note that TCS also could have been used, but this tool is no longer maintained, because of insufficient resources. It was used for the extension of ATL however, because ATL is still written using TCS. Extending the syntax would take more time if it first had to be ported to XText.

```
1  grammar nl.sytematic.dsl.AnnotationDSL
2      with org.eclipse.xtext.common.Terminals
```

```
3
4  generate annotationDSL "http://annotationDSL"
5
6  Root :
7          (annotations+=Annotation)*
8  ;
9
10
11 Annotation :
12         metaClass=ID '[' (selectors+=Selector)* ']' '{'
13                 (features+=ID)*
14         '}'
15 ;
16
17 Selector :
18         'name' '=' val=STRING
19 ;
```

Listing 5.10: The grammar of the annotation model.

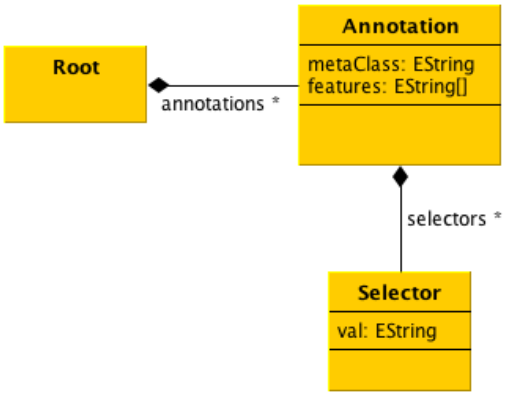

Figure 5.8: Annotation meta model, generated by XText

5.6 Conclusions

This chapter addressed the problem of handling variability in a product line that is implemented using a transformation sequence. It stated that normal ATL rules are able to handle variability, but not without problems (ie. imperative code, changing the transformation after the feature selection has changed). Therefore the approach was to create variability management means by introducing a new type of rule called variability rules. The motivation for adding a new type of rule was the observation that rules are purpose-built to do mappings, thus they are a good candidate to do the required mapping between feature model and meta models. Also, rules provide mature mechanisms for matching, querying, and instantiating meta model elements.

Our concept achieved at least three things. First, the implicit execution order is maintained, yielding the same modularity properties as normal rules. Second, all the advanced mapping features of normal ATL can be used, as well

as the rationale that is behind conventional rules; the developer can use the same rationale (of matching elements) for variability rules as for normal rules. Third, because it is a higher-order transformation, the ATL engine is not modified, so there is not a new ATL version that has to be installed (and maintained!). Instead, one could just chain the HOT into the transformation sequence.

We have shown ways to solve two types of variability: source model independent, and source model dependent variability. For the latter, there are some open issues left, the main one being that there is not yet a way to check if a selection is valid. This check can no longer be outsourced to the feature model editor because it needs information from the source model. A simple annotation model with a convenient, CSS-like, concrete syntax was developed, which keeps the source model from being cluttered by annotations.

6 Usage Guidelines for Variability Rules

6.1 Introduction

This chapter explains how to use variability rules in practice. Section 6.2 shows three potential situations in where to use or not to use variability rules. Section 6.3 onwards describes a step-by-step guide in setting up a transformation sequence which can handle variability efficiently. These sections cover all the artefacts, like feature models, meta models, annotation models and transformation definitions, that play a role in the transformation sequence.

6.2 Situations in where to use or not to use variability rules

With a reusable HOT implemented, this section shows how variability rules can help managing both types of variability in three conceptual situations. In every scenario, we consider a transformation sequence of *m-1* transformation steps, as shown in Figure 6.1. For each scenario step $n < m$ is considered, and it is explained if and how variability rules are useful. The source of each transformation could be a manually created model or a generated one. For either one the following theory will apply.

All scenario's are stated in terms of the *step the feature realization artefacts of a certain feature are introduced in the transformation sequence.*

We abbreviate meta model with 'MM'.

6.2.1 Scenario 1 — Feature realization artefacts introduced in MM n

In scenario 1, the first notion of a certain feature x in terms of realization artefacts is in MM n, as Figure 6.2 shows.

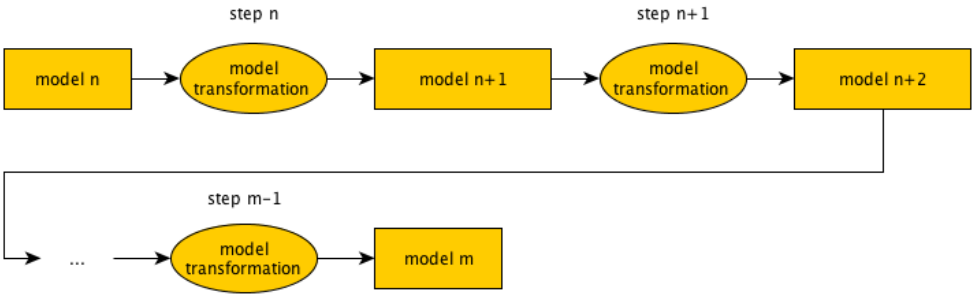

Figure 6.1: A transformation sequence with m models, or $m-1$ transformation steps.

In the scenario displayed in Figure 6.2, variability is handled by variability rules. In the source meta model MM n the variation point p is introduced, and is specialized to variant x using a variability rule.

A variability rule for x and for y will be present in the ATL transformation, where the *to*-clauses are different, creating x- respectively y-related realization artefacts depending on the feature model. The parent feature p of the source model acts as a point for a normal ATL rule to match upon in the *from*-clause.

Listing 6.1 shows an example of the use of variability rules in scenario 1.

Note that if the source model had x or y instantiated (instead of the ParentFeature p), one could use normal rules to match on x or y. In that case, it be compared to Scenario 3, below.

```
1  variability rule x configures BaseRule {
2      from
3          a : MMn ! p --parent feature of x & y
4      to
5          x_a : MMn+1 ! x.a (
6                    x_b <- x_b,
7                    x_c <- x_c
8          ),
9          x_b: MMn+1 ! x.b (
10                   x_c <- x_c
11         ),
12         x_c : MMn+1 ! x.c
13 }
14
15 variability rule y configures BaseRule {
16     from
17         a: MMn ! p  --parent feature of x & y
18     to
19         ...
20 }
```

Listing 6.1: Example variability rule for feature introduction in MM n.

Similarly, but more concretely, consider the following example, in Listing 6.2, for the running case transformation sequence. Here we have a feature group *DataBaseTechnology* in the source meta model, with two variants, *RDBMS* or *XML*. This means that the first occurrence of the feature is in the source

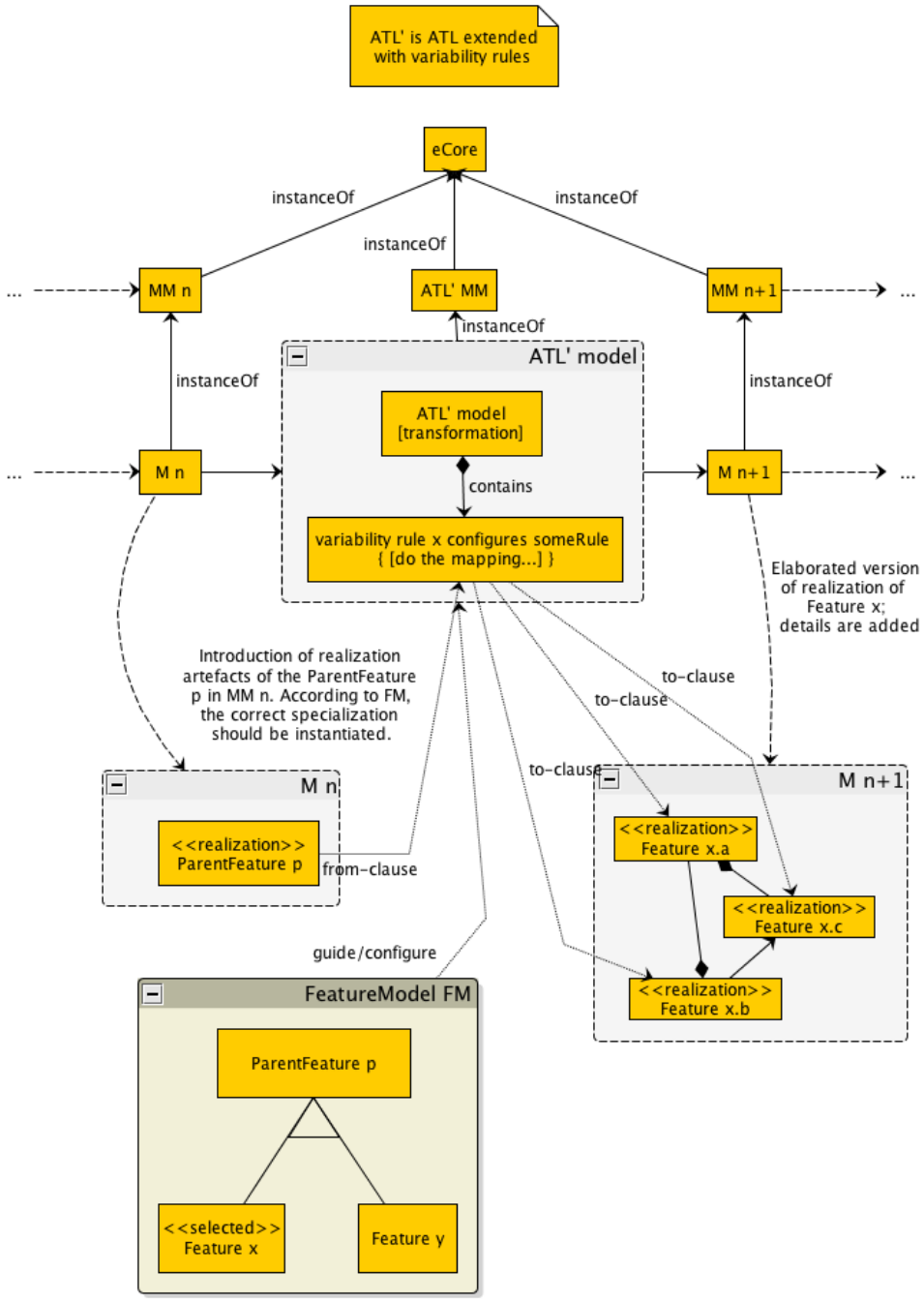

Figure 6.2: Usage scenario 1: Feature realization artifact(s) introduced in MM n.

meta model, which is the concept of Scenario 1. Note that, as with the previous example, if the *from*-clause of the rules would have been like $a : MMn!RDBMS$

or $a : MMn!XML$, one could have used normal rules, as it is then already clear from the source model which feature should be integrated. In this case, the source model just states: 'there is a DataBaseTechnology', and the transformation definition's task is to instantiate the correct specialization, according to the feature model.

```
1  variability rule RDBMS configures DB2DBSpecialization {
2      from
3          a : MMn ! DataBaseTechnology
4      to
5          db : MMn+1 ! RDBMS
6  }
7
8  variability rule XML configures DB2DBSpecialization {
9      from
10         a: MMn ! DataBaseTechnology
11     to
12         db : MMn+1 ! XML
13 }
```

Listing 6.2: Concrete example variability rule for feature introduction in MM n.

6.2.2 Scenario 2 — Feature realization artefacts introduced in MM n+1

In scenario 2, the source meta model has no notion of the variation point *ParentFeature* or the feature x or y. This scenario is closely related to the previous one, but here the *from*-clause is matching anything from MM n, ie. the source meta model does not match on feature realization artefacts like in Listing 6.2.

Strictly speaking, scenario 1 is not different from this one, except that the *from*-clause matches on a feature realization artefact, and scenario 2 matches on anything from MM n. Scenario 1 is put here in the first place for completeness. However, it might make sense to explicitly model the concern in the source meta model, as in Listing 6.1 and 6.2, for instance because of model clarity. Another reason might be that this yields the possibility to do its configuration in the feature model, while still having a notion of the grouping feature in the source model.

As we shall state later more elaborately, scenario 2 is the preferred one. In fact, our transformation sequence case only uses variability rules where the situation is like scenario 2. This means that in the target meta model of a transformation step the first notion of a particular feature realization artefact occurs. So as an example, look at the variability rules for database technology in Listing 5.4

6.2.3 Scenario 3 — Feature realization artefacts introduced in MM n–1

To cover all the grounds, we demonstrate the case of a feature realization that has been introduced before, in MM n–1, shown in Figure 6.4. Here, normal rules are used for the mapping. Using normal rules is appropriate, because the fact

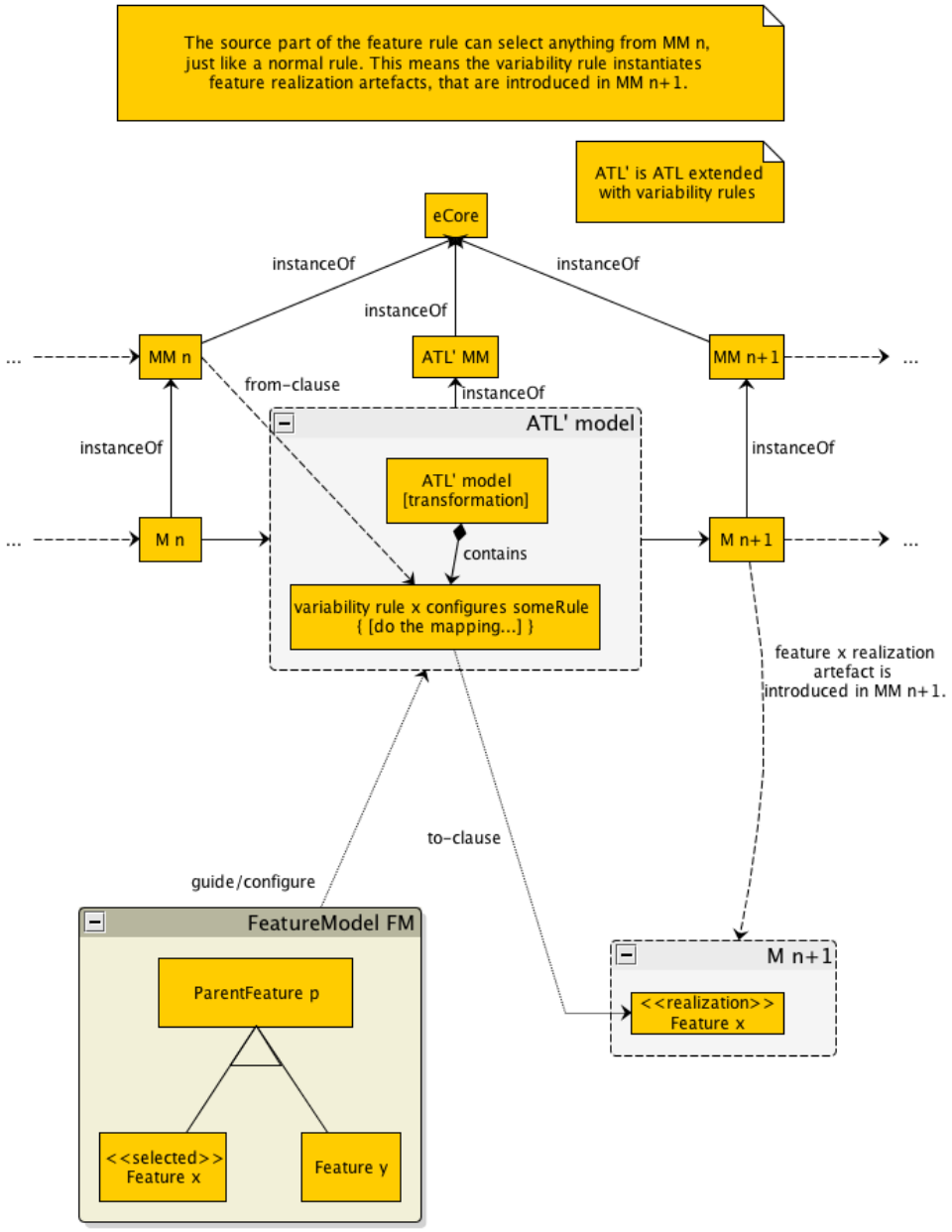

Figure 6.3: Usage scenario 2: Feature realization artifact(s) introduced in MM n+1.

that feature x is selected can be inferred by the existence of an instance of it in the source model. One can match on x in the *from*-clause without the chance of running into the problem of having to match multiple rules for a single source element. The feature configuration concern is out of the equation in scenario 3; it has already taken place in an earlier step.

The next section gives our recommendation on what scenario is preferable, and what rationale should be applied in a transformation sequence design.

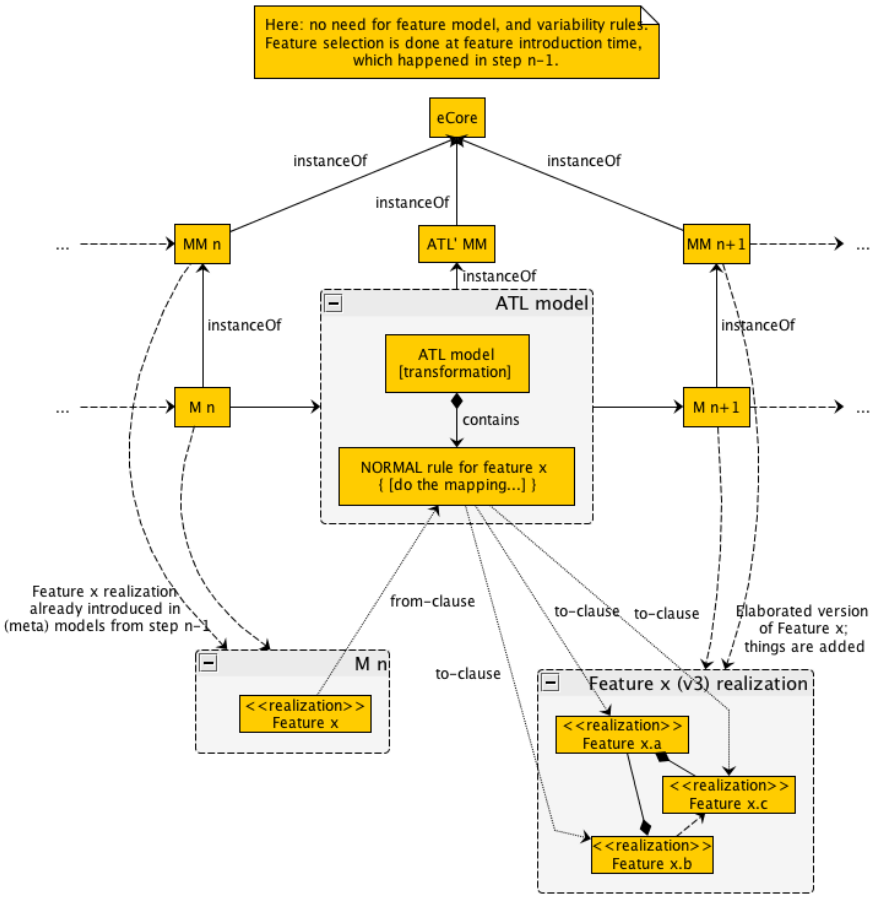

Figure 6.4: Usage scenario 3: Feature realization artifact(s) introduced in previous meta model MM n-1. Using normal rules is appropriate, because the fact that feature x is selected can be inferred by the existence of an instance of it in the source model.

6.2.4 Recommended scenario

As a **best practice guideline**, having something like scenario 2 (Figure 6.3) is recommended. Then, feature realization artefacts are introduced in the target meta model of a transformation, and the variability rules instantiate the features (ie. create instances of these realization meta artefacts) in their *to*-clause. In steps that follow, one could just use normal rules to match on the realization artefacts from the source meta model and refine it, like in scenario 3 (Figure 6.4). The fact that a feature is selected, and thus introduced following scenario 2 in a previous step, can be inferred from the very existence of an instance of a feature realization artefact in the source model.

As a consequence, variability rules are only used when necessary, and normal rules are used as much as possible. The variability rules will now act as an integrator/instantiator of feature realization artefacts. We prefer to use normal rules in steps that follow, because the rules will now be regular refinement mappings, and the concern of feature selection is now taken out of the equation in these steps (as in scenario 3). Only at introduction-time variability rules are used.

The final result is that one can just change selections in the feature model, or make annotations in the annotation model, and the transformation sequence will generate a system integrating features following the feature diagram and the annotation model.

6.3 Setting up a transformation sequence to handle variability

Like many other language constructs, variability rules are only effective if used right. This section describes a set of guidelines that can be used to achieve the best result.

The following sections explain the design decisions one should take, when defining a step-wise refining transformation sequence with variability management.

This chapter considers a transformation sequence that already has the common parts integrated. The variable parts now have to be added to the transformation definition, and the next sections show how.

6.4 Step 1 — Model variability in feature model

Whenever a new type of variability is identified, one should add it to the feature model. This gives insight in the constraints that a certain feature will have. Now, one can categorize the variants in terms of source model (in)dependence variability. In the dependent case, one needs to add annotations in a later phase.

6.5 Step 2 — Model variability in meta models

As mentioned earlier, variability rules are meant to work with the feature realization artefacts residing in meta models. Therefore, one has to model the realizations in the meta models. First, one has to decide on which step in the transformation sequence, and thus in which meta model, the feature realization should be introduced. Some features are of a higher abstraction level than others. The higher the abstraction, the earlier it will be introduced.

When features are of the type 'alternatives' or 'one or many', one should consider modeling them with inheritance or a comparable construct. This is a convenient way, and it will allow a simple one-to-one mapping with variability rules. In other words, the variability rule definition will be more modular and thus more reusable and adaptable.

6.6 Step 3 — Define variability rules for each feature

For both cases, source model independent variability and source model dependent variability, one is encouraged to write the variability rules in the transformation definition for which the target meta model is the model that introduces the feature. In other words, one is encouraged to follow the situation shown in Figure 6.3.

6.6.1 Case A: independent variability

In this case, there will be at least one variability rule for each feature. One has to determine according to which rationale the feature should be instantiated. For instance, if the source meta model is ECore, it could be for each EClass, or for each EAttribute. Thus, one has to determine the relevant source meta element to be matched. Usually, there is already a rule with the same *from*-clause. This rule can be used as base rule.

Since we consider the independent case, the transformation definition for this feature is finished.

If a single feature has to be integrated for multiple source elements (ie. for each EAttribute and for each EReference), one can create multiple variability rules with the same name, differing on the base rule they configure. Then, one would have something like shown in Listing 6.3.

```
1  variability rule FeatureX configures
2                  EAttribute2SomethingElse {
3          from a : ecore!EAttribute
4          to ...
5  }
6
7  variability rule FeatureX configures
8                 EReference2SomethingElse {
9          from a: ecore!EReference
10         to ...
11 }
```

Listing 6.3: Integrating a single feature according to multiple source elements.

Typically, the decisions on 'what to match' depend on the way the meta models is structured. One can use the same thought process as used in normal ATL, with normal ATL rules.

6.6.2 Case B: Dependent variability

As far as matching goes, the rationale is the same as with independent variability rules. However, since the integration of a certain feature can differ on a per-source element basis, the feature should be modeled in the target meta model in a certain way. Concretely, one should make sure the situation is similar as shown in Figure 6.5.

Here, each common element A is generated from an element $srcA$ by a certain rule, say rule X (see Listing 6.4). In this situation, each A has a reference r to one of the feature specializations $\{F1, F2, F3\}$. Thus, each feature will be

integrated for each *srcA*, and bound to each *A* eventually (if it is annotated with a feature).

This means it is *not possible* to instantiate a dependent feature that is *not connected via a reference* to an element from the *to*-clause of the configured base rule. Thus we have the following requirement on the target meta model.

Target meta model requirement: In the target meta model it is required to have a reference *s* from an element created by a base rule *B* to a dependent feature *F*. Then, the variability rule instantiating this feature should configure *B.s*.

```
1 rule X {
2         from s : sourceMM!srcA
3         to t : targetMM!A
4 }
```

Listing 6.4: Base rule transforming the common element srcA into A.

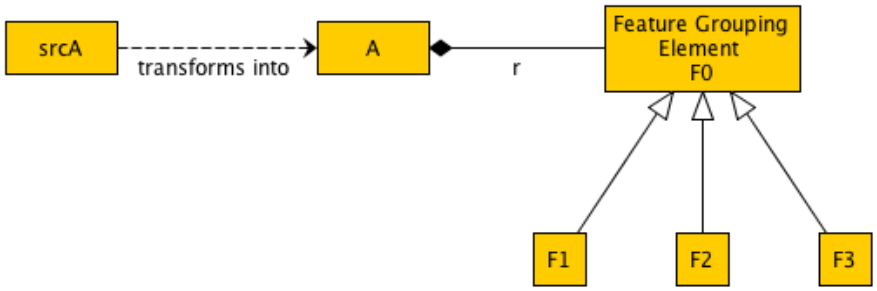

Figure 6.5: Preferred structure of target meta model with source model dependent variability.

With this structure a dependent variability rule will be easy to create. In our situation this rule will look as Listing 6.5. Note that the variable name of the feature is the same as the reference: *r*. This is obligatory, since the HOT will resolve the binding using the first variable name of the *to*-clause.

It is of course possible that the feature realization artefacts consist of more elements. One should try to keep one root element for every feature realization subgraph, and put it as the first element in the *to*-clause of the variability rule. This element has outgoing reference edges, and only one incoming edge (*r* in this case). If this is obeyed, one can create multiple elements in the *to*-clause of the variability rule, and this can be linked accordingly. Other references should be resolved using the *resolveTemp*() method of ATL.

```
1 variability rule F1 configures X.t {
2         from s : sourceMM!srcA
3         to r : targetMM!F1
4 }
```

Listing 6.5: Dependent variability rule for feature F1.

The question could be asked: 'isn't this too restrictive?'. We don't think so. When creating a target meta model in which a set of dependent features is introduced, one knows on which source element a feature depends. That is, one knows which source element should be annotated. It is natural to have a base rule, like in listing 6.4, which generates a target element (A) from this source element ($srcA$). The only requirement, then, is to have a reference between this target element and the features.

We cannot think of situations where having this link gives a problem. If there are any, then this is the limitation of the current version of dependent variability rules.

6.7 Step 4 — Define annotations in an annotation model

With everything in its place, one can add annotations on elements of the source models. One annotation model drives a certain configuration (in combination with a feature model). If a different configuration has to be made, either the feature model has to be changed, or the annotation model.

6.8 Conclusions

This architecture can manage variability, and facilitates the automated integration of variable parts into end products. The annotation model and feature model are the drivers. Next to that, the concerns are separated in different models. The variability rules are declarative, with implicit rule ordering.

The next chapter gives an evaluation of what was achieved, focusing on the most important new asset: variability rules.

7 Evaluation

7.1 Introduction

This chapter looks back at what was achieved, and puts it in perspective by comparing it to other approaches. The key result was the introduction of variability rules in ATL. Therefore we first assess the quality characteristics of this approach. After that, the additional required models, feature models and annotation models, are discussed.

7.2 Quality Properties of Variability Rules

The quality assessment can be done over different aspects of the variability rules. The assessment is based on the following quality characteristics: expressiveness, modularity, maintainability and usability, performance, reusability and adaptability. These value of these quality properties can be quite subjective, therefore we compare it to normal ATL first. If variability rules match the quality characteristics of normal ATL rules, we consider it successful.

The quality properties *reusability* and *adaptability* are put in perspective with Kurtev et al.'s [KvdBJ07, Kur05] work.

7.2.1 Expressiveness

Variability rules match normal rules to a great extent when it comes to declarative expressiveness. However, the current implementation does not support the *using*-clause and *do*-clause. For the running case, we did not need these construct, and they are left as future work. It should not be hard to implement.

Furthermore, OCL expressions can be used just like in normal rules. The only thing that is not possible anymore is putting OCL guards (ie. OCL expressions that return a boolean) in the *from*-clause of variability rules to prevent a rule from calling. However, guards are not needed, because the semantics of

variability rules determines whether a rule should be called or not, determined by the feature model and, in the source model dependent variability case, the annotation model. Thus, if variability rules are used as intended, the expressiveness matches the normal rules to a great extent.

As far as the semantics is concerned – call a rule if-and-only-if the corresponding feature is selected – a transformation developer can now achieve variability management by simple declarative variability rules. This is an advantage over using normal rules and imperative code, where one has to do tedious work to create a transformation with the same semantics. Thus, the expressiveness of ATL' is better than normal ATL.

7.2.2 Modularity

Variability rules share characteristics with sub-rules which extend a base rule. In fact, variability rules are inspired by them. Like sub-rules, variability rules can use all the declared variables in the base rule. Furthermore, they are non-invasive; the base rule is oblivious to what sub-rules it has and what variability rules configure the base rule. So, as far as modularity goes, there is no difference between rule inheritance and variability rules.

As far as the HOT is concerned, the HOT is a modular and reusable component, with well-defined in- and outputs. It can just be chained in a transformation sequence as a preprocessor.

7.2.3 Maintainability and Usability

Variability rules are as declarative as normal rules, so they are as easy to understand and use as normal rules. Furthermore, because the ATL' compiler is written as a higher-order transformation, one does not have to maintain two ATL engines. In fact, users of the HOT can benefit from updates to the ATL engine. This makes it easier to maintain. After the HOT has run, the resulting ATL model can be used for further processing, so one can use the normal ATL debugger to step through code. Also, one can see the results of the HOT preprocessor, and see what it compiled to. The resulting code is human-readable, improving usability.

Furthermore, the same philosophy of matching source elements have to be used with variability rules, just like normal rules. Therefore the same paradigm of matching source elements and creating target elements *for each* of these matched elements, applies. Therefore, next to the fact that the syntax is not very different to normal rules, makes it relatively easy to learn.

These two properties are hard to judge on how successful we were in obtaining them. This thesis will not make a more thorough attempt.

7.2.4 Performance

Running a higher-order transformation might be slower than integrating the variability rule code generation in the ATL compiler. Yet, we think that the benefits on the other fronts makes it a small price to pay. In the real-life product line case described throughout our thesis, it is not a show stopping disadvantage. We trade better variability management for eventual lower performance in transformation execution. Also, running the HOT takes less than a tenth

of a second on our computers, whereas model-to-text transformation form the bottleneck, because they have to create a lot of files, which is a slow process. Thus, performance is not an issue in our case.

7.2.5 Reusability and Adaptability

As Kurtev et al. [KvdBJ07, Kur05] have pointed out, reusability and adaptability of transformation rules are two quality properties which are important. Here we assess how well variability rules are doing.

Transformation definitions can now be modular along the variability dimension, which was not possible before. Specifically, the instantiation of variants – the features – can be modularized, improving reusability.

However, recall that variability rules can be, as far as coupling goes, compared to rule inheritance. According to Kurtev et al., rule inheritance has some limitations concerning adaptability. They claim that rule inheritance introduces tight coupling to the base rule.

This being true, we think that it is still a good solution. The reason lies in the nature of variability. Typically, parts of a common rule (ie. elements of the *to*-clause of the base rule) are variable. These variable parts (ie. the feature realization artefacts) in the meta models are often conveniently modeled via an inheritance relation. Having an inheritance-like relation in the transformation language allows for a straight-forward, one-to-one mapping between the variability rules and the features in the meta models.

If the base rule changes, the variability rule has to change, which is a downside. The upside of the inheritance-like relation between variability rules and base rules, is the fact that variability rules can use the variables from the base rule. Note that this is especially useful if the target meta model also uses an inheritance relationship between the variation points and the variants, that is, the feature group and the possible features.

As a solution to the coupling problem, the *configures*-clause of an (independent) variability rule could be made optional. Then, there is no shared scope of variables of the base rule, but the advantage is that the variability rule is not coupled to anything. For some features that are loosely coupled in the meta models, this might be useful. The optional *configures*-clause is not implemented in the current version, and is left for future research.

7.2.6 Conclusions of the quality properties

To conclude this section we claim that on the reusability quality of the complete transformation definition, variability rules bring a real benefit. The variability rules add means for handling variable aspects.

Furthermore, because of the declarative nature, the rules are more simple, and no imperative code is needed, making the rules shorter and easier to comprehend. Simple rules influence the actual need for reusability.

As a final remark, quality properties have effect on each other, and we do not claim to provide a quantified assessment on the values of these quality properties. We *do* claim that variability rules are a step forward; a number of quality characteristics is improved.

7.3 Evaluation of the Feature models and Annotation models

Here we evaluate the other artefacts. There are two models needed: feature models, and annotation models, for which we discuss the pros and cons of their usage.

7.3.1 Feature model

For the source model independent case, the results are satisfactory, because only a feature model is needed. A feature model can be created by any feature model editor, provided it has an injector that can load a textual or graphical representation into an EMFModel object.

The advantage is that the configuration – and its constraints! – can be outsourced to a feature model editor, which can take the responsibility of checking the validity of the feature model, and prevent the user from making forbidden selections. This outsourcing means that we do not have to worry about the configuration semantics at all, but just focus on *what to do if a feature is selected*!

7.3.2 Annotation model

On the source model dependent variability case, there is no way around some sort of annotation. There are multiple places where annotations can be put: in the input model, in the feature model, or as a separate model. Having a separate annotation model is a benefit, in our opinion, because of at least four reasons.

Firstly, it does not clutter the input model with annotations. Secondly, the input model is oblivious that it is being annotated, improving modularity and separation of concerns. Thirdly, the convenient and simple syntax of the annotation model can be compared to Cascading Style Sheets (CSS), which is comparable as far as usability goes. Fourthly, putting the annotations in the feature model, in some way, compromises the reusability of the feature model, because the annotations need to be removed for every new domain model, thus having them in a separate model is beneficial.

7.3.3 Checking feature selection validity for source model dependent variability

Features that have to be instantiated on a per-class or per-attribute basis, ie. the source model dependent variabilities, have a fundamental problem. When they are put in a feature model, it is not clear what is a valid feature selection and what not.

If a feature model only contains source model **in**dependent variability, its feature selection can be validated by an algorithm (often integrated in the feature model editor). The algorithm cannot validate source model dependent feature selections, since there is no such thing as 'a feature selection'; a feature can be 'selected' for one instance of class A in the input model, but 'deselected' for a different instance of class A. Instead, a feature selection being 'valid' now also depends on the annotation model, and this should be checked by the HOT.

However, for the feature selection validity check we first should define what is 'valid' and what is not. We propose the following semantics, explained by an example.

Consider a target meta model where a class *Controller* has a reference to one or more *Views*. The class *View* has two sub-classes: *GridView* and *TreeView*. The feature model *FM*, states that *GridView* and *TreeView* should be alternatives (ie. *xor*). The Controller-View combination elements are created *for each EClass* element of the source model. The variability rules will look like Listing 7.1.

```
1  --Base rule
2  rule EClass2Controller {
3          from
4              a: inMM ! EClass
5          to
6              c: outMM ! Controller
7  }
8
9  variability rule GridView configures
10                 EClass2Model.c {
11     from
12         a : inMM ! EClass
13     to
14         views: outMM ! GridView (
15             name <- a.name+'TreeView'
16         )
17 }
18
19 variability rule TreeView configures
20                 EClass2Model.c {
21     from
22         a : inMM ! EClass
23     to
24         views: outMM ! TreeView (
25          name <- a.name+'TreeView'
26         )
27 }
```

Listing 7.1: Feature constraints from the feature model operate on a per-Controller basis in this example due to the *.c* part of the *configures*-clause pointing at a *Controller* element.

We now state that the *View* features should be alternatives per-*Controller*, due to the fact that the *configures*-clause points at a *Controller* (defined in the *.c*-suffix). This means that for an EClass instance *A*, the transformation creates a *Controller* with a *TreeView xor GridView*. A *Controller* generated from a different EClass instance *B*, can also have a *GridView xor TreeView*. So, for example, if a *Controller C* has both a *TreeView and* a *GridView*, this is invalid.

In the running example case, this seems the most reasonable semantics. The check is not implemented, and is left as future work.

7.4 Limitations and future improvements

The implementation that is provided here can be regarded as a proof-of-concept, so there is room for improvement. The following list shows the recommended

improvements:

- Currently, only *EClass*es can be annotated.

  The HOT's $isAnnotatedWith(...)$ method uses a hardcoded EClass as parameter. This method should be re-implemented with a reflective nature, to be able to check if an element of any type is annotated with a certain feature.

- Make variability rules possible without the *configures*-clause for looser coupling

  There might be feature realization artefacts that have loose coupling in the target meta models to common parts, that is, features that have no dependencies on other elements and are very easy to compose. For those features it might make sense to have variability rules without the *configures*-clause pointing to a base rule. Then, such a rule will have no coupling to a base rule, improving reusability. This is not yet implemented, for the reason that the variability in the running case *did* benefit from the inheritance-like relation of configuring a base rule, due to the fact that the scope of the variables was shared.

- In the current version, only classes can be matched to create new elements in the target model.

  If one wants to use variability rules on attributes or references, this is not possible. However, this is also not possible in normal ATL (ie. one cannot create a rule that only creates an attribute, it also has to create a first-class element like an EClass, after which the attribute can be set). In most cases this is not a problem, since most meta models have first-class elements for things like references and attributes, to be more flexible. In the case of Ecore, there are classes EReference and EAttribute. In the CRUD system meta model they are called ModelReference and ModelAttribute, respectively. Variability rules work well in such a case.

- The XText parser that parses annotation models is currently not very mature.

  It is now capable of annotating only by selecting on a *name* attribute. Probably it would be nice to allow other attributes, or even full OCL expressions. Such a feature has impact on the HOT, since the $isAnnotatedWith(...)$ helper that is generated need to be more generic. This could be made generic by reflection.

- The infrastructure should make sure that all features that are used in an annotation model actually exist in the feature model.

  This could be implemented in the annotation model parser, or in the HOT. Currently, the developer needs to manually check if all features are spelled correctly in the annotation model.

- The HOT should check if a source model dependent feature selection is valid.

  It should do this by querying the annotation model, and assessing if the set of annotations is valid, according to the semantics described in Section 7.3.

- Industrialize the whole package, that is:
  - Minimize the amount of configuration needed for a user that wants to use the HOT,
  - Include a package with an easy-to-use injector, which parses a ATL' file, and injects it into a runtime EMFModel object. Currently this parser is in a separate Eclipse project,
  - ATL' projects should easily be created in Eclipse, complete with a parser and an injector.

7.5 Comparison to other Approaches

There are other approaches that need to manage variability, albeit in different contexts. This section lists some of them, and compares them to the solution from this thesis, as far as that is possible. Some approaches are not solving the core problem that we wanted to solve, but reason about other parts of the infrastructure. Therefore they are also included here.

7.5.1 Variability Management in a Model-Driven Software Product Line

The work of Garces et al. [GPA+07] comes close to both the goals of this research, and the way they implemented it. Like in our research, they want to manage variability in model-driven product lines, hence the corresponding title. In their approach, they also create custom rules that are meant to manage variability from a feature model. Their rules are called (i) base rules, (ii) control rules and (iii) specific rules.

The difference with the approach here, is that their rules are implemented in ATL itself. They propose a pattern of combining these rules in a specific way to handle variability. So control rules are in fact rules that have controller functionality that decide whether a feature has to be called. This feature, then, is implemented in a specific rule, which is a called rule. The control rule decides if the specific rule is to be called.

In other words, Garces et al. propose an ATL software pattern, a specific structuring of ATL rules, which has to handle the variability.

The disadvantage of their approach is that the ATL definitions become more complex, and involve more imperative code. Their advantage is that no HOT is needed, just normal ATL will do. Furthermore, our approach is more declarative, and abstracts the user from the semantics of integrating a set of features.

To conclude, while their research goal corresponds with ours, we think that our solution yields model transformation definitions with better quality properties (as defined before).

7.5.2 Weaving Variability into Domain Metamodels

According to Morin et al. [MPL+09], there is variability in Domain-Specific Modeling Languages (DSMLs). DSMLs can describe a wide range of different models. These models often share a common base, and vary on some parts. Variability is often managed using a separate variability language, which they

find a bigger learning curve for DSML stakeholders. Furthermore, they see it as a significant overhead in the development of product lines for DSMLs. They consider Model-Driven DSMLs, where the product line and also the derived products are models.

They proposed to consider variability as an aspect to be woven into the DSML, that is, into the domain meta models, to introduce variability capabilities. It is a reusable variability aspect, defined at the meta model level. The variability concepts and their relationships are described independently from any domain meta model. The result is that an architect can model artefacts from the domain with variability included.

Morin et al. generate modeling languages with variability management capabilities, whereas here, the variability handling and configuration happens in the transformation sequence itself.

7.5.3 Variability within Modeling Language Definitions

There is a considerable quantity of variability mechanisms in use by modeling languages today. The approach of Cengarle et al.[CGR09], attempted to formalize the semantics of modeling languages, in particular UML 2. The objective was to specify variability management formally. These formal semantics can be used in any object based modeling language. In other words, their work is about variability *within* a language, ie. in the development of the language itself, *not* the variability that can be *expressed* using the generated language.

Their solution is a taxonomy of variability mechanisms. In the context of modeling language definitions, there can be three kinds of variability: *i*) Presentation, *ii*) Syntactic and *iii*) Semantic variability. Next to this taxonomy they provide a framework to manage and document the variation points in a modeling language. Their approach is to specify a modeling language using MultiCore, which generates an abstract syntax tree (AST). This specification includes a model of the concept of variability.

The method they used in dealing with variability, in this case in modeling languages, is based on traditional code generation. Because this use of traditional methods, there is no concept of source model dependent and independent variability. This means that variability is expressed differently, not like in this thesis which opts for a declarative approach.

7.5.4 Leveraging model transformations by means of annotation models

Vara et al. [VBVM09] described a way of annotating models which act as a driver for configuring a model. This work inspired me to use annotation models, rather than clutter the input model with annotations. Their approach is used in the context of databases, and schema generations/transformation.

Like their approach, our approach uses helpers in the ATL transformation that check if something is annotated. In their approach, this is a manual process, whereas in our case, the annotation check is now woven into the transformation definition by the HOT. Our approach results in more declarative code as with is possible with their approach. So, their concept is the same as ours, but we raise the level of abstraction a bit further.

7.5.5 Aspect-Oriented Model-Driven Software Product Line Engineering

According to Voelter and Groher [GV09], effectiveness of a software product line depends on how well feature variability is managed in the whole development cycle. They also realize that aspect-oriented software development and model-driven software development are different, but can complement each other. Their idea is to integrate these two practices for the sake of variability management capabilities.

Their approach is to use the openArchitectureWare toolset to define feature models, that will be woven with solution space models, done by aspect weaving. This approach allows to have a separate, orthogonal feature model that configures the input model. On multiple levels in the transformation sequence, aspects are used to modularize crosscutting variabilities. A case study of a SmartHome is used to demonstrate the approach.

This project shows an interesting, thorough approach for managing variability throughout a model-driven software product line. Aspect-orientation is used in models, in model-to-model transformation code, and in model-to-text transformation code. Consequently, a developer has the freedom to implement the variability it encountered –as specified by the feature model– on multiple levels of the transformation sequence.

The difference with this approach is that they do not use ATL as the model-to-model transformation language, thus they do not solve the problem for ATL users. Furthermore, the approach in this thesis does not use aspect orientation, but standard model element matching and new semantics. In other words, our solution provides a rule based mapping mechanism between feature models and feature realization artefacts in meta models. We have a different method of managing variability, with as advantage that the same philosophy as normal rules is used, lowering the learning curve for developers.

Most importantly, their problem can also be solved by our variability rules.

7.5.6 Traceability between Feature Model and Software Architecture

Satyananda et al.[SLKH07] discuss the problem of identifying traces between a feature model and software architectural artifacts from the problem domain. They argue, that this can be done manually for small cases, but an automated approach would be useful in larger systems. Their approach is based on Formal Concept Analysis. First a functional decomposition from both the feature description and architectural description is (automatically) extracted. This is possible due to a functional decomposition language construct in the definition of both models. After that, FCA uses these functional decompositions and the feature model and the architectural model as input, analyzes it, and returns a mapping between feature model and architectural model. The mapping is created using a graph-searching algorithm.

The problem they try to solve is a relevant one, also in the field of MDE. In the end, a feature model should be connected to artifacts from the transformation sequence. However, their approach is purely PLE based, and does not speak about the concepts of model transformations. With our MDE approach, the traces are made available by the model transformation. Therefore, the MDE

approach solves this issue, that is relevant for PLE, advertising the use of MDE technologies in the creation of product lines.

7.5.7 FeatureMapper: Mapping Features to Models

In PLE, feature models are often used to manage variability, and to derive products from a common set of artifacts. To be effective, a mapping is needed between features and solution space artifacts, which allows for automatic product derivation. Heidenrich et al.[HKW08] approached this problem by building a tool called FeatureMapper, that maps features from a feature model to software artifacts, but also interprets and visualizes these mappings. It support automatic mapping, where the mapping is made while the developer is modeling solution space models. The tool provides a record-mode for this, which records what the developer is modeling. Finally, the tool can interpret a mapping to create models that only include the parts that are needed for a given variant. This transformation can be started by the tool, but also comes as an openArchitectureWare workflow component.

The FeatureMapper would in our case map features to feature realization artefacts from the meta models. Thus it would do the same as ATL does in our case, if one obeys Scenario 2 (see Section 6.2), where the feature realization artefacts are residing in the target meta model of a transformation. However, ATL is perfectly capable of doing this, and also provide OCL expressions. Thus the FeatureMapper would not be advantageous over the use of ATL in our case. The FeatureMapper could be used to manage source model dependent variability, but we prefer to use the CSS like syntax of the annotation model. This is not intrusive, and declarative, and textual, although this makes the FeatureMapper not necessarily a less suitable approach of mapping elements to features.

7.5.8 Using Feature diagrams with Context Variability to model Multiple Product Lines for Software Supply Chains

Hartmann and Trew [HT08] discuss the use of feature diagrams to model multiple product lines. Often, there is contextual variability in multiple product lines. Therefore they introduce the concept of Context Variability. They illustrate it with examples from the automotive industry, for a car infotainment system. The context model here consists of the region a car is sold, the price class of the car, etc. The context variability model constrains the feature model, enabling the possibility to model multiple product lines supporting several dimensions in the context space.

The relation to their work is that the transformation sequence of their project can potentially generate multiple product lines. This is because it receives input at the meta model level; each input model is defined as a meta model. This means the domain changes per system generated with the transformation sequence. The *input meta model independent variability* (shown in Figure 4.1), as it is called in our report, can be regarded as Context Variability, the term used in [HT08]. However, since in this thesis there is no need for multiple product lines, this is not implemented.

7.6 Conclusions

In this evaluation chapter, we explained what was achieved, and put it in perspective against other approaches. Also, the variability rules implementation was evaluated. We stated that the implementation works as a proof-of-concept, but should be industrialized more. Specifically, some editors need to be more mature, with mechanisms like constraint validation built-in, and the $isAnnotatedWith(...)$ method of the HOT should be re-implemented using reflection, such that any meta model element type can be fed to it. Industrializing also means adding an ATL' editor and injector, that is more mature as the current version.

Another improvement would be to make the *configures*-clause optional. In some cases, features that are loosely coupled (or not coupled at all) do not necessarily need to be instantiated or integrated *for each* meta element. This, however, needs some experimentation, and a more thorough study. Rules without a *configures*-clause would be less coupled, and thus more modular, due to not being dependent on a base rule.

To conclude, the proof-of-concept is usable, but there is room for improvement, on the tool-support aspect.

8 Conclusion

This last chapter formulates the conclusion of this thesis. First a summary is given of the thesis, focusing on what is delivered, and what is the quality of the delivered product. The section following the summary answers the research questions.

8.1 Summary

The motivation for the research in our thesis started by the observation that the model-driven product line case lacked in the management of variability. Specifically, there was a need for configuring the individual applications (ie. family members) that were derived by the product line. It was chosen to make the model transformation language ATL capable of dealing with variability in the case of developing applications by step-wise refinement. The reason was that ATL would be a good candidate to provide mappings between features from a feature model and feature realizations artefacts from the meta models. What's more, ATL is well-known for creating target elements from source elements in a declarative way, but lacked good language constructs for handling variability. Therefore, we followed the ATL philosophy; create a language construct modeled after an ATL rule, which can match its quality properties. These quality properties consists of expressiveness, modularity, implicit execution order.

The concept is called **variability rules** which follow the semantics of sub-rules. Like sub-rules, variability rules reuse a base rule, but the difference is that they are only called when a corresponding feature is selected in the feature model, for the source model independent variability case. However, sub-rules suffered the problem of being unable to match one source element by multiple sub-rules. Variability rules's semantics is to overcome this problem, which was necessary, since potentially multiple features should be integrated *for each* particular type of source element.

The initial variability rules could not cope with source model dependent

features, features that can be applied on a per-class basis. Therefore the syntax was extended further, to be able to specify what to do if a class from the source model is annotated with a certain feature.

So, in the case of source model dependent variability, there is also information needed from the input model (ie. the source model of the transformation). This information is provided by annotating the source model. To prevent cluttering of the source model, a separate annotation model was created, which decorates a domain model. Its syntax is modeled after Cascading Style Sheets, the popular language for decorating HTML files.

The implementation of a compiler that compiles an ATL' file, that is a file with ATL rules and variability rules, was done using a higher-order transformation (HOT), written in ATL. The HOT compiles an ATL' model back into an ATL model, with only selected featured enabled. The resulting ATL model can be used for further processing, so the HOT acts as a preprocessor. A major advantage of using a HOT is that the normal ATL engine is not touched. Thus it benefits from update to the ATL engine.

It was shown that variability rules are best used in situations where a feature is introduced in the *target* meta model of the transformation. The other situations can be handled with normal ATL rules, although it is theoretically possible to use variability rules in the scenario where the feature is introduced in the source meta model. It is however, not recommended, since the fact that it exists in the source model means that it is selected in the feature model.

8.2 Answers to the Research Questions

The following section looks back at the original research questions, and states how they are answered. These research questions were distilled after observing some limitations of the initial version of the transformation sequence. The research questions were made as general as possible, yielding results that can be used in other contexts. Answering the sub-research questions should yield an answer to the overall research question.

The first sub-question was: *What kinds of variability are apparent in the example case?*

This can be answered by stating that there are two types of relevant variability, source model independent variability, and source model dependent. The first one needs only information from the feature model to be handled properly, whilst the latter also needs an annotation model. The dependent case is more complex by nature. Consequently, the variability rule syntax is more elaborate. It also means that the feature model editor can no longer be the one to check validity of a feature selection. The check has still to be done, by the higher-order transformation for example, which is left as future work.

The following questions follows the former one:

How can the MDE infrastructure be adapted to manage these types of variability?

The infrastructure was adapted by adding three things: a separate feature model, a separate annotation model and variability rules, a language construct added to ATL, yielding ATL'. Our approach achieved separations of concerns, modularity, which increases maintainability and usability. The declarative nature of variability rules, which follows the philosophy of normal rules means

there is not a steep learning curve. It also means that there is no need for imperative code, something which is discouraged by the ATL philosophy.

The final question's answer is partially stated in the previous paragraph, and was the following:

What are the limitations of ATL when it comes to variability management, and how can these be overcome?

The limitations of ATL were in the first place in the inability to match multiple rules for a single source element. This key feature was provided by variability rules. These rules use a feature model and an annotation model to declaratively state what feature realization artefacts from the target meta model should be instantiated and integrated in the target model. This solves the ATL limitations rather nicely, since all the quality characteristics of normal rules are maintained: modularity, declarative code, implicit execution order, and the full set of OCL expressions are available.

These questions answered the main, global research question:

"'How to manage variability in MDE-based product lines that derive products by step-wise refinement using the ATL model transformation language?.'

This means that the final answer was that there should be a separate feature model, a separate annotation model, and a modified version of ATL, called ATL', which features variability rules. The combination allowed a modular, declarative solution, with the concerns of configuration and annotation separated from the transformation code. Due to the nature of the source model dependent variability type, there was the need for annotation, which meant there had to be made a compromise to the ease and separation of feature configuration; no longer is the feature model the only driver for selecting features. However, the solution has as characteristics that the annotation, is non-invasive, uses a simple syntax, and is truly separate from the rest of the architecture. Consequently, the other models are oblivious to the existence of the annotation model.

8.2.1 Applications of this Framework in other contexts

The example transformation sequence described in this thesis was the motivation for most of the solutions provided. This section assesses the extent to which it can be applied in other situations.

The solution, that is, the combination of feature model, annotation model and variability rules, can be applied in other situations. Now follows a list which states some situations:

- Any step-wise refining transformation sequence with the need for configuration.

  Meaning that as soon as there is variability to be integrated in the output of the transformation sequence, one can provide a solution using the framework of our thesis. One should model the variability in a feature model, encode the way the features should be integrated in variability rules, and depending on the nature of the variability, should annotate the input model.

- Translating models to different platform specific models.

  A common use of MDE is to create multiple platform specific models (PSM) from an platform independent model (PIM), which could be used

to generate for instance a Java application and a C#.NET application from a common PIM.

Rather than having two ATL transformations, one for each platform, one could use variability rules. With two ATL transformations, there is a potential code scattering. Then, only the ones that are relevant for a particular platform could be called, by configuring the feature model. Not only is the integration of features beneficial, but also the fact that the variable parts are modeled in a separate model, which increases the understanding of the differences between platforms.

In this situation, mostly source model independent features are likely to exist, rather than dependent features. This means that an annotation model might not be necessary.

A potential problem of this strategy could be (as opposed to having two ATL transformations), that the target meta model contains realization artefacts of both platforms, but this depends on the size and complexity. On the other hand, ATL allows for multiple target models, so it can be overcome.

8.3 Future Work

Some improvements could still be made. First, source model dependent features are not checked for validity, because extra information is needed from outside the feature model. In the classical case, and also in the source model independent variability case, the feature model could be validated on its own. That is, no other information is needed to validate a feature model only consisting of source model independent features. When source model dependent features are included, there is also the need of the input diagram, and the annotation model. Then, following self-defined semantics, a check should be done. This is left as future work.

Other future work includes: improving the parsers, and industrializing the proof-of-concept, by making it easier to create ATL' projects and run them.

8.4 Final Remarks

When some of the future work would be completed, our infrastructure is a useful framework for creating product lines in a model-driven way. However, it is already used in practice by our case, so the current status is already usable. Industrializing the package will make it easier for other people to use it, which hopefully leads to a growing user base. While our information system case is further developed in the near future, we try to improve the tools alongside with it.

The appendix contains all the source-code and models on a CD-rom, and is a good place to get started.

References

[ACR] H. Arboleda, R. Casallas, and J.C. Royer. Dealing with Constraints during a Feature Configuration Process in a Model-Driven Software Product Line. *DSM'07*, pages 178–183.

[AK03] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20:36–41, 2003.

[BBM03] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[Béz05] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Béz06] J. Bézivin. Model driven engineering: An emerging technical space. In *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2006.

[BPSP04] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[BSR03] D. Batory, J. Neal Sarvela, and A. Rauschmayer. Scaling step-wise refinement, 2003.

[CE00] K Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CGR09] M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within modeling language definitions. In A. Schürr and B. Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer, 2009.

[CSS] Cascading style sheets, http://www.w3.org/style/css/.

[Fav04] J. Favre. Towards a basic theory to model model driven engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.

[GPA+07] K. Garces, C. Parra, H. Arboleda, A. Yie, and R. Casallas. Variability management in a model-driven software product line. In *Avances en Sistemas e Informática*, volume 4 No. 2, pages 3–12, Sept 2007.

[GV09] I. Groher and M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. *Lecture Notes in Computer Science*, 5560:111–+, 2009.

[Hib] Hibernate orm framework, http://www.hibernate.org/.

[HKW08] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: mapping features to models. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 943–944, New York, NY, USA, 2008. ACM.

[hs10] IDI Software http://www.idi software.com/resources/defns.html. Definition variability management, 04 2010.

[HT08] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society.

[IKV] IKV++Technologies. Mediniqvt, http://www.ikv.de.

[JBK06] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[JK06] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[Ken02] S. Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[Kur05] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, ISBN 90-365-2184-X, 2005, 2005.

[KvdBJ07] I. Kurtev, K. van den Berg, and F. Jouault. Rule-based modularization in model transformation languages illustrated with atl. *Science of Computer Programming*, 68(3):138 – 154, 2007. Special Issue on Model Transformation.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MOFa] Meta object facility 1.4, http://www.omg.org/technology/documents/formal/mof.htm.

[MOFb] Meta object facility 2.0, http://www.omg.org/spec/mof/2.0/.

[MPL+09] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. Jézéquel. Weaving variability into domain metamodels. *Model Driven Engineering Languages and Systems*, pages 690–705, 2009.

[oAW] oAW. Xtext http://www.eclipse.org/xtext/.

[PBvdL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 1 edition, 2005.

[Sij10] M. Sijtema. Introducing variability rules in atl for managing variability in mde-based product lines. In *MtATL Workshop 2010*, 2010.

[SLKH07] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi. Identifying traceability between feature model and software architecture in software product line using formal concept analysis. In *ICCSA '07: Proceedings of the The 2007 International Conference Computational Science and its Applications*, pages 380–388, Washington, DC, USA, 2007. IEEE Computer Society.

[SQL] Mysql to km3, http://www.eclipse.org/m2m/atl/atltransformations/.

[TCJ10] M. Tisi, J. Cabot, and F. Jouault. Improving higher-order transformations support in atl. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin / Heidelberg, 2010.

[TJF+09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg, 2009.

[VBVM09] J. M. Vara, V. A. Bollati, B. Vela, and E. Marcos. Leveraging model transformations by means of annotation models. In F. Jouault, editor, *MtATL*, pages 96–102, 2009.

[VG07] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

[xml] exstensible markup language, http://www.w3.org/xml/.

[YZZ+07] X. Yu, Y. Zhang, T. Zhang, L. Wang, J. Hu, J. Zhao, and X. Li. A model-driven development framework for enterprise web services. *Information Systems Frontiers*, 9:391–409, 2007. 10.1007/s10796-007-9042-7.

A

Appendix: Compact Disc

Accompanying this report is a CD containing all the files that are needed to replicate the case. This appendix explains the directory structure.

On the root level, there are three directories:

- **/Code**

  This contains the code, consisting of models, meta models, editors and transformation definitions. In this directory there are seven sub directories.

  - **/1. input_model**

    Contains the input (meta) model, used to generate the system shown on Figure 3.2 in chapter 3. It is an instance of Ecore.

  - **/2. feature_model**

    Contains the feature model from the case, as well as the meta model, and an Eclipse Genmodel project to generate an editor plugin.

  - **/3. annotation_model**

    Contains the annotation model from the case, as well as the meta model, and an XText project to generate an editor plugin.

  - **/4. ATL-prime**

    Contains the ATL' meta model, a model (.fatl) and an injected version (.xmi), and a TCS project to inject and extract ATL' models, and an editor. The term featureATL is used in filenames as ATL'.

  - **/5. ATL-prime_to_ATL_HOT**

    Contains the Eclipse project for the HOT.

  - **/6. Ecore_to_CRUD**

    Contains an Eclipse ATL project for transforming a HOT-processed transformation, which transforms an Ecore model, with an annotation model as guide, into a CRUD model.

  - **/7. CRUD_model**
    Contains the CRUD meta model, called gwt.ecore.

- **/Presentation**

  Contains the presentation slides used for the collocquium. It was created with Apple Keynote, but also a Microsoft Powerpoint export is included. This may have influence on the visual effects. Also note that the slides are not self-describing, they require a presenter in order to make sense.

- **/Report**

  Contains the report, as well as the LaTeX source files and images. For all the diagrams, the .graphml source files are also included. These can be edited by yEd, amongst other programs (or one could write an injector, and make them available as EMF representation for automated processing!).