TU Clausthal

Thomas Ternité

Variability of Development Models

An approach for the adaptation of development models

SSE-Dissertation 3

Software Systems Engineering

Department of Informatics
Chair of Prof. Dr. Andreas Rausch

Variability of Development Models

An approach for the adaptation of development models

D o c t o r a l T h e s i s
(D i s s e r t a t i o n)

to be awarded the degree of
Doctor rerum naturalium
(Dr. rer. nat.)

submitted by
Thomas Ternité
from Mainz-Mombach

approved by the Department of Informatics,
Clausthal University of Technology

2010

Dissertation Clausthal, SSE-Dissertation 3, 2010

Chairperson of the Board of Examiners
Prof. Dr. Michael Kolonko

Chief Reviewer
Prof. Dr. Andreas Rausch

2. Reviewer
Prof. Dr. Stefan Biffl

Date of oral examination: November 5, 2010

Cover picture: ©iStockphoto.com/marekuliasz

Dedicated to Dr. med. Hella R. Schäfer

Abstract

Introduction of process models (development models) is a common approach for organizations to aim for a better process quality and higher success rates in project conduction.
In the context of this thesis, the notion of 'process models' will be extended to a notion of 'development models'. This is due to the fact that in the common sense of process models, they are usually not only related to processes, but to other artifacts, as well.
Prior to introduction of development models into an organization, a comprehensive customization is usually inevitable.
Along with an intended customization, the following questions are relevant:

What concepts can be used to...

- ... support the customization in general?
- ... minimize the customization effort?
- ... assure conformity of the customized variant to the intentions of the original model?
- ... keep the customized variant in line with a developing original model?

There are different approaches implemented in actual development models like the V-Model XT and RUP to address these issues. Although they are effective in the details, they lack a holistic approach.
This thesis offers an analysis of variability mechanisms in development models, software product lines, and other software/model related domains. The findings are integrated into a framework for variable development models. In addition to offering variability, this framework allows the definition of domain-specific and model-specific constraints that are used to discard models that do not structurally conform to the intentions of the original model's creator.
When adequately realizing this framework, a variable development model is understood as a *development model line* (DML). A DML is offered to an organization's process engineer as a means to create variants of the original model. While doing this, he uses explicit variability mechanisms, namely configurability, extensibility, and modifiability.
The target audience of this thesis is the engineer of development models. This includes both the engineer of an original (standard) model and the customizing engineer of an adapting organization.

Acknowledgment / Danksagung

An dieser Stelle möchte ich mich bei den Menschen bedanken, die mich auf dem Weg unterstützt haben, diese Arbeit abzuschließen.
An erster Stelle danke ich Prof. Dr. Andreas Rausch für die einmalige Gelegenheit, gestalterisch herausfordernde und zugleich technisch anspruchsvolle Tätigkeiten im Bereich des Software Systems Engineering ausüben zu dürfen. Sein kritischer Geist und sein Blick für unstimmige konzeptuelle Details in allen Fragen der Wissenschaft und der praktischen Anwendung sind mir in den letzten Jahren ein großes Vorbild gewesen. Danke auch für die Intensität der vielen Diskussionen zum Thema dieser Arbeit.
Mein Dank geht auch an Prof. Biffl und seine Mitarbeiter für das herzliche Willkommen in Wien und das Feedback zu meiner Arbeit.
Ich danke Hella Schäfer für die liebevolle, vollumfängliche Unterstützung in allen Belangen meines Lebens. Ihrer Familie spreche ich ebenfalls meinen Dank für die Aufnahme in ihre Reihen aus, die mir seit Jahren eine große Hilfe ist.
Meinen Eltern möchte ich für den Rückhalt während meines Studiums und danach danken. Danke auch für das vermittelte Gefühl eures Stolzes.
Ich danke Sabine und Dirk Niebuhr für die ganze erfreuliche Zeit, die wir miteinander verbringen, und für das reichhaltige und sehr willkommene Feedback zu kritischen sowie unkritischen Inhalten dieser Dissertation.
Danken möchte ich auch Jan Friedrich, Marco Kuhrmann und Klaus Bergner für die vielen fruchtenden Diskussionen beim Redesign des V-Modell XT Metamodells, dessen Neuerungen letztlich auch Gegenstand dieser Arbeit sind.
Meinen mittlerweile zahlreichen Kollegen möchte ich dafür danken, dass wir es trotz starkem Zuwachs irgendwie geschafft haben, eine Gemeinschaft zu bleiben. Ich hoffe, ihr könnt diese Gruppenintegrität noch lange aufrechterhalten.

Thomas Ternité,
Clausthal-Zellerfeld, den 11. August 2010

Contents

List of Figures

List of Tables

List of Definitions

Preface

I am convinced that a native association with language makes any text easier to understand for the reader. So I use 'I' if my person is of relevance.[1] Verbal excesses like "The author points out..." or usage of 'we' if I actually mean 'me' simply won't happen.
I will refer to us, i.e. 'we', if it is convenient to assume that you, the reader, are putting yourself notionally into the presentation of my thoughts and arguments. A sentence like "Imagine that elephant we discussed on page 22, but this time with five legs" is allowing for a more fluent reading and is generally more appealing than expressions like "In this section, the elephant presented on page 22 is extended by a fifth leg".
So, to state this clearly: 'we' will never be me pretending to be a vague group of scientists, but always you and me together on our path to a better understanding of the subject.
In case it's not self-evident in the Table of Contents, I offer a short description of the anatomy of this thesis.
In addition to a List of Figures and a List of Tables, this thesis provides a List of Definitions that contains all definitions that are relevant to the concept presented in this document.
The thesis is subdivided into three parts. The second part covers the solution concepts. The other parts serve as a preparation and recapitulation of these concepts.
The appendix contains code used for an exact specification of semantics that are needed for the concepts. After these appendices, a bibliography is provided. It's followed by two indices.
The first index lists the entities that are relevant in the main concept presented in PART II. The second index is a general index covering the relevant keywords throughout the whole thesis.

[1] Though, this is not the case in any chapter of this thesis.

Part I.

The Domain of Development Model Adaptation

Contents of the First Part

1. Introduction

Contents

In software and systems engineering, we observe that methods, technologies, and processes play an important role in respect to the quality of the developed systems. Methods and technologies represent the set of tools to engineer systems. There are a lot of different approaches for software and systems engineering that fill in methods and technologies for a large variety of use cases.
In respect to processes, and according to the Chaos report published by the Standish Group, common reasons for development project success are: proper planning, clear statement of requirements, clear objectives, user involvement, executive management support, and small project milestones [Sta94].[1]
Defined processes are a way to ameliorate these criteria for a development project. The usage of development models is a common approach for projects that shall realize software and systems within a predefined budget, time, and quality [FK07].
Development models are focused on describing processes, and on combining these with methods and technologies for usage in development projects.
Standard development models are usually very generic to allow a universal applicability. This universality is often a problem in respect to the applicability of a standard model. This implies the adaptation of such a model to an organization's processes and structures. A standard development model rarely fits well to the requirements of an applying organization.
Thus, an organization's aim is often to provide more concrete methods and a guideline to project managers. In addition, processes usually have to be adapted to better embed the projects into the process framework present in the organization.
In larger organizations, a development model customization is a very complex task [AEH+08], even without the introduction process [Arm08]. Not only the model usually has an increased complexity after customization, but it is difficult to maintain a model and keep it in line with the original model, if it evolves in parallel.

[1] The validity of the Standish report's figures has been challenged [Gla06, EV10], but the influence of these success factors is commonly undoubted.

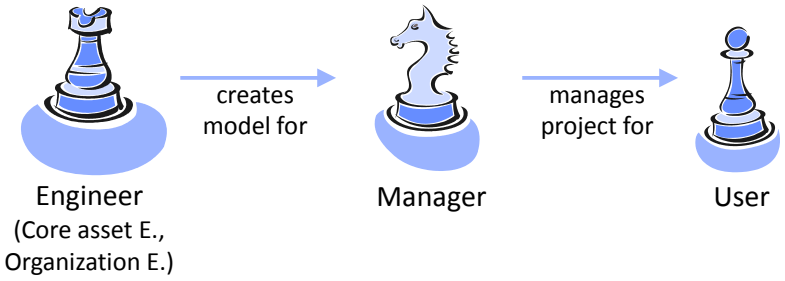

Figure 1.1: Roles around a development model

An example for such an evolving model is the V-Model XT. Since its release in 2005, six new versions of the model were released, mainly due to feedback from users.[2] For an organization with a customized variant of any release, it is difficult to keep in touch with the evolving standard model.
In short, these difficulties motivated the creation of this thesis.
In the area of development models, research is divided into three parts according to the supported roles. It affects:

- the user of a development model,
- the manager of a project as a special case of a user, and
- the engineer of a development model.

Firstly, the enactment of development models can be supported in respect to methods and technologies, to enable a holistic application during a development project. This subsumes all users of a development model, shown on the right hand side of FIGURE 1.1.
Secondly, the project manager instantiates guidelines provided by a development model. He fills the gap between process descriptions, the actual project, and its overall project plan. How to remain consistent in this case may be within the focus of research.
The last direction concerns the definition of development models. That's were this thesis is focused on. It is intended to provide a concept for variable development models. These variable development models are to be constrained in a way that not all imaginable variants are within the range of possible development models, but only those that are admissible to certain conditions. The engineer can either be focused on a development model standard (core asset engineer), or on an organization-specific development model (organization engineer).
The target audience of this thesis is the engineer. It provides support in respect to variability during creation of a development model (target group: core asset engineers) and during adaptation of a development model (target group: organization engineers).

[2] Namely, the release versions after 1.0 were 1.01, 1.1, 1.2, 1.2.1, 1.2.1.1, and 1.3.

1.1. Research issues

With the focus being placed upon the engineering of development models, this thesis contributes to the *development model definition process*. The following five goals will frame the contents of this thesis.

Goal 1 *To investigate how 'variability' is commonly understood during development of variable development models and software.*

The notion of the term 'variability' for development models and software is perceived in different ways in the literature and in practice. It is a goal of this thesis to give a structure to this term.
In short, variability of development models or software is usually perceived as a property to be configurable, extensible, modifiable, or a combination of these three attributes.
The following goals relate to development models only, as this thesis concentrates on the variability of development models, not software. Variability of software is taken into consideration by GOAL 1 because we can observe variability mechanisms there that are important for the variability approach presented in this thesis.

Goal 2 *To create a pattern concept for variability of development models, regarding the notion of variability as it is identified in* GOAL 1.

The definition of the term *variability* as targeted by GOAL 1 is complemented by the identification of specific patterns that enable variability of development models. These patterns can be used to realize configurability, extensibility, and modifiability of development models.

Goal 3 *To identify the aspects to be regarded when constraining the variant space of development models.*

In order to not only create arbitrary variants of development models, this thesis identifies the aspects to be regarded when constraining the variant space of development models.
This is realized by a discussion on different mechanisms for variant restriction.

Goal 4 *To combine the variability patterns (targeted by* GOAL 2*) into an architectural framework for variable development models, namely a development model line framework.*

This goal covers the construction of an architectural framework for variable development models, using variability patterns with properties identified in GOAL 2. The framework can be used for the creation of variable development models. Such models represent a base model for a complete family of development models. This family is called a *development model line*.

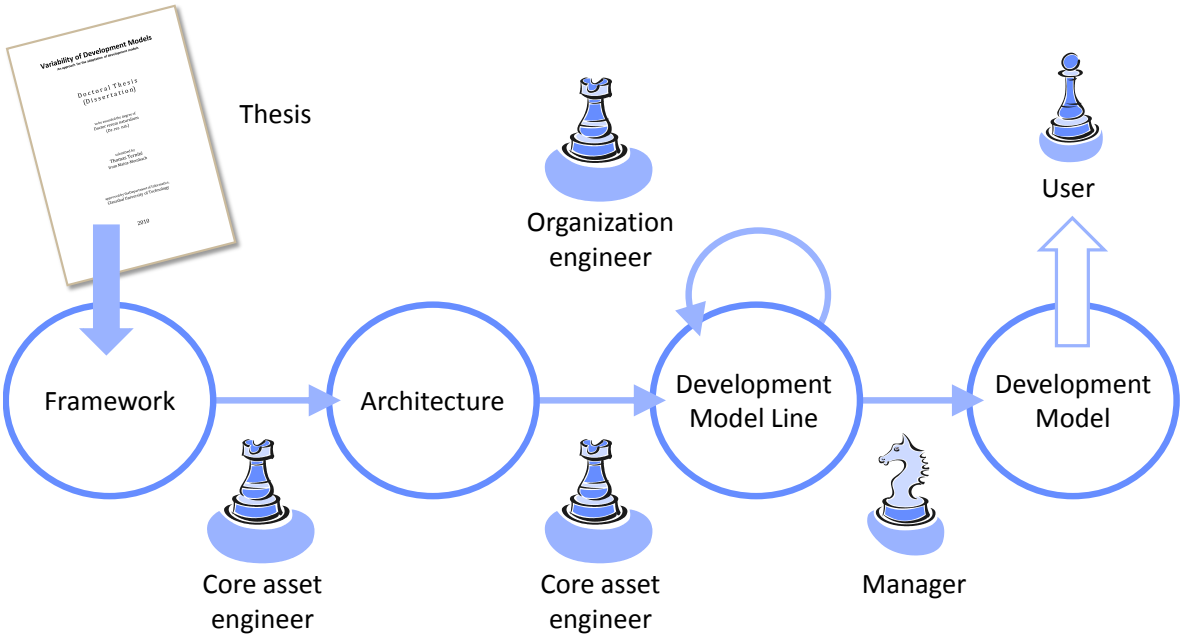

Figure 1.2: Chosen solution approach

Goal 5 *To integrate a variant restriction mechanism into the development model line framework.*

With the knowledge of variant restriction mechanisms (GOAL 3), a concept will be presented that is intended to ensure that variable development models fitting to GOAL 4 are restricted to an admissible variant space.

1.2. Solution approach

It is an aim of this thesis to provide a solution used to achieve the goals described in the previous section. The thesis proposes a concept for support of the adaptation of development models to organization-specific and project-specific needs. The concept includes variabilities to be used by engineers of organization-specific development models, as well as by project managers wanting to adapt a development model to a particular project.
FIGURE 1.2 shows the structural constituents of this concept. This thesis proposes a framework that is responsible for providing variability possibilities. This framework is intended as a basis for the creation of concrete development model architectures. This creation is performed by a model engineer developing a standard development model. In effect, an architecture is the meta-model of a development model, i.e. it defines the model element types that can be defined within a development model.

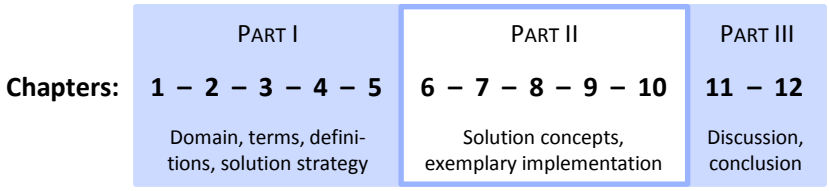

Figure 1.3: Thesis structure

A core asset engineer then uses the architecture to model a development model line. A development model line is a set of development model related assets. These that can be configured by a project manager to be used as a development model. This development model is adopted by the user in a project.
This configuration is supported explicitly by the concept. Thus, the project manager performing this step can rely on existing contents and tools enabling a simple tailoring process.
In addition to its configurability, a development model line is extensible and modifiable. These properties are important to an organization engineer creating a customized development model line for an organization.
An existing implementation that is largely consistent with the notion of a development model line that is configurable, extensible, and modifiable is the V-Model XT [BMI08]. This thesis makes use of the very same concepts for these variabilities like the V-Model XT and its tool implementation, the V-Model XT Editor [VMX09].
Nevertheless, one contribution is the addition of a generic layer above the metamodel of the V-Model XT and its tool implementation. This layer is targeted by GOAL 4. Another contribution is the addition of a formalized mechanism for variant restriction, which is targeted by GOAL 5.

1.3. Thesis structure

This thesis consists of three parts. Their structure is depicted by FIGURE 1.3.
The first part contains all definitions relevant in the other parts. It introduces the domain of variability and the adaptation of development models (CHAPTERS 2–4). In addition, with CHAPTER 5 it provides a detailed description of the solution approach that was shortly described in the previous SECTION 1.2.
PART II represents the conceptual emphasis of this thesis. In CHAPTERS 6–9, it describes a concept realizing the solution approach framed by CHAPTER 5. CHAPTER 10 describes an exemplary implementation of these concepts.
This implementation was primarily used for evaluation of the evolving concept in respect to soundness, consistency, and correctness. It does only provide a minor case study for the application of the concepts in respect to applicability and scalability.
Finally, the last part contains a discussion of the proposed concept (CHAPTER 11) and a conclusion (CHAPTER 12).

2. Adaptation of development models

Contents

2.1. Development models

Development models are used to organize a development process. A common way to denote the subject of this thesis, namely development models, is the usage of the term ‘process model’. As we will see below, in this thesis’ notions of this domain, a process model is a constituent of a development model.
The definition stated in this thesis will not stick to a purely intensional definition only. Instead, this thesis makes use of both intensional and extensional descriptions.
In general, an intensional definition describes a set of attributes that fit to all elements that are subject to the definition. For development models, there is no intension known describing all items that are commonly understood as development model, without allowing more things that are commonly understood as not being a development model.

On the other hand, an extension is describing all elements belonging to the notion of development models by naming each of them. As there is no full list of all development models, nor can we foretell the names of all development models yet to come, a complete extension of development models cannot be written down. In addition, a pure extensional definition is a weak definition, as it prohibits the upcoming of new items to be subject to the definition.
This thesis contains an intensional definition of the term 'development model', but knowing that this definition must be incomplete and inaccurate, it is augmented by an additional extensional description of its properties.

Definition 1 *(**Development model (intension)**) A development model is used in development projects for the creation of software and systems and has the following properties:*

- ***Purpose:** Its purpose is to reproducibly increase the **success rate** of development projects.*
- ***Methodology:** A development model is a **knowledge pool** to be used as a guideline or instruction reference during development projects. As a knowledge pool, a development model contains sub-models that describe various **reference areas** to map the ideal model[1] to a real development process, in order to be able to adapt the reality towards this ideal. These reference areas are used to give answers to questions concerning the creation of the resulting product.*

A project is deemed to be successful if the resulting product is of a high quality,[2] and the budget and time frames both stay within predefined limits [Sta94].
The definition above is incomplete. It does not provide any suggestions on how a development model may succeed in fulfilling its purpose as a knowledge pool used for the successful accomplishment of projects. In order to reduce, not eliminate, this flaw, an additional extensional description of these properties is provided. The given lists are incomplete as well, but they span the area as it is known today.

Definition 2 *(**Development model properties (extension)**)*

- *Known approaches to successfully execute development projects are:*
  - ***Development paradigms** to work towards a higher product quality: code-and-fix programming, stagewise development, iterative development, incremental development prototyping, V-shaped models, transformation based models, agile models, and reuse oriented models.*
  - ***Management disciplines** to stay within a predefined allowance of quality, budget and time: project management, quality management, risk management, configuration management, and change management.*

[1] Note that 'ideal' does not imply 'optimal'. The term bears a notion of a good exemplar, though.
[2] The more features and functions implemented as initially specified, the higher the quality.

- ***Reference areas*** *are used as an ideal description of how the reality should be formed according to the development model. Such descriptions can be grouped into product model, role model, activity model, process model, location model, and rationale model. These are often referred to as 'sub-models'.*

The following sections describe these properties in more detail.

2.1.1. Approaches for successful project execution

This section contains a list of methods and strategies that were developed to increase the probability that a project finally is successful. The approaches are sorted into two subsections: one for approaches intended for the organization of development processes (development paradigms), and one for cross-functional approaches to increase the quality of the resulting product and to control time and budget of project execution (management principles).

Development paradigms

All of the following approaches have one thing in common: they were developed with the aim to enhance the quality of software and systems by following a concrete development paradigm.
It is very common to combine the following development approaches with each other, creating new variants by making use of different strategies all at once. Illustrations of many such variants have been provided by FORSBERG and MOOZ [FM01].

Code-and-fix programming. The code-and-fix model, known from the earliest days of software development [Boe88, p. 61], is an intuitive approach for software development. Quality is achieved by abidingly following two development steps, again and again:

1. Write some code.
2. Fix the problems in the code.

This approach may appear trivial, but it's a concept making the difference between coding and testing explicit. This is creating a more transparent development process.

Stagewise development. This approach is organizing a development process into a sequential order with clearly defined milestones and quality goals for all work products to be created until a milestone is finally reached [Ben87, p. 305]. A prominent example is the waterfall model, which additionally allows feedback loops between stages [Roy87, p. 330] as an additional mechanism to enhance quality.
Many of the following approaches are making use of this stagewise development principle, embedding it into a more complex flow of activities.

Iterative development. Iterative development is a concept making use of the observable fact that a single development cycle rarely is sufficient to produce a complete software system with an appropriate quality. It relies on several cycles, while each cycle is running through several phases of a stagewise development. A famous development model explicitly introducing this paradigm is BOEHM's spiral model of software development and enhancement [Boe88, p. 64]. Iterative development is characterized by repetitions of development activities [Coc93, p. 311]. The main focus of these repetitions is rework, not development of new items.

Incremental development. While iterative development is making use of repetitions of development activities to do rework, in an incremental approach, repetitions of development activities are prevalent, too. But in case of incremental development, the activities address new parts of the system, instead of reworking existing ones [Coc93, p. 311]. I.e., incremental development focuses on the addition of parts to the specifications and the system (an increment) during each repetition, i.e. the system is developed in portions.
A description of what can be the focus or character of an increment, e.g. the implementation of stub implements, was provided by PITTMAN [Pit93, p. 51].
Incremental development may be combined with an iterative approach. In this case, each single increment may be developed using an iterative development, i.e. each increment is reworked until it reaches shipping quality [Coc93, p. 312].

Prototyping. Stagewise development has drawbacks [MJ82] that led to the formulation of an evolutionary development model [Boe88, p. 63]. This model more directly involves the future user of the developed system in the development process. The software is developed iteratively or incrementally, while the user evaluates the result of each iteration. A redesign of the prototype and its specification is made according to the user's input.
Evolutionary development is one case where a prototype is created that is refined and evaluated step by step.
Another use of a prototype can be explorative, where a prototype is created merely to have a medium to show to one or more user representatives and talk about the requirements. I.e., explorative prototyping is a method for requirements elicitation.
Prototyping can also be used in an experimental setting to evaluate different solution possibilities for problems emerging from the requirements specification.
With explorative and experimental prototyping, the prototype is usually discarded when its purpose is fulfilled. The now emerging need to 'cleanly' reprogram the system is intended to increase its quality by ruling out dirty code refactoring.

V-shaped models. A V-shaped model is explicitly using verification and validation at exactly defined steps throughout a development process [Nat07]. During specification refinement, it is always verified that a finer specification is consistently covering the relevant parts of the original specification. Verification is intended

to assure that on the path from requirements specifications, over system and architecture design, down to module design, the resulting module specifications are consistent with the requirements. This part has been visualized by FORSBERG and MOOZ as the left bar of a 'V' [FM92].
On the right bar, rising from bottom to the top right of the 'V', the realization of the modules, and the integration to units and systems, up to the complete system, is validated against the specifications at the appropriate level on the left bar. This means that modules are tested against the module specification, a system is tested against its system specification and the whole system is tested at user-level against the requirements specification.

Transformation based models. In contrast to the other approaches listed above, this approach does not involve the ordering of process steps. Instead, it is a paradigm that relies on automated generation of code using a formal specification [BCG83]. Its intention is to increase the quality of software even if the software was adapted several times in the maintenance phase. Modifications are made to the specifications, not to the code.
Product quality is increased by avoidance of code decay during development and maintenance.

Agile models. An agile process is usually characterized by self-organization of small teams, direct communication, frequent re-evaluation of plans, and short iterations. The 'Agile Manifesto', formulated and signed by many members of the agile community [BBvB+01], identifies four main prioritizations:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

While the authors of the Agile Manifesto agree that there is value in the items on the right, they perceive the items on the left as being more important.
Widespread agile processes are eXtreme Programming [BF01], Feature Driven Development [PF02] and Scrum [SB01].

Reuse oriented models. Building systems from existing components is an approach to both save development time and to rely on successful and tested assets [Sam01]. The system is created by selection, acquisition, and customizing existing components from one or more component repositories [AF98]. In addition, some specific modules are created in a way that the existing components may interact appropriately.
Software product lines are realizing this concept [CN02].

Management disciplines

In a project, there usually is a lot management of activities and work items to be done. Studies show that efficient management structures affect critical success factors for project execution [FW06, p. 55 f.].
In general, these supporting disciplines can be divided into project management, quality assurance management, risk management, change management, and configuration management.
Maturity models are used for the evaluation of process maturity. They usually describe requirements for management disciplines a development model must fulfill in order to enable successful project execution.
ISO/IEC 15504 (SPICE) is an international standard for the assessment of enterprises with a focus on software development [ISO06]. Capability Maturity Model Integration (CMMI) is a series of reference models that can be used as a frame to compare a concrete development model with [SEI06].
Both maturity models describe the assessment and evaluation of projects and organizations concerning supporting disciplines like project management. They are regarding quality aspects, too, and SPICE makes use of aspects of the software lifecycle,[3] as its nomenclature is oriented on ISO/IEC 12207 (Software Life Cycle Processes) [ISO95].

Project management In the German standard DIN 69901, project management is described as an entity of organization and executive functions, with appropriate methods and instruments [DIN01]. Project management is used to coordinate the time line of projects, in respect to delivery milestones, as well as to the available budget [HHMS04, p. 6].
Project management covers project organization, controlled project startup and closing, project planning, as well as project examination and control [HHMS04].
PRINCE2 is a governmental standard model for project management widely spread in the UK [OGC02]. It is described in SECTION 2.1.3.

Quality management Quality management in general is about quality control, quality assurance and quality improvement. I.e., it covers not only the quality of an end product (quality control), but also the systematic monitoring and evaluation of processes (quality assurance), as well as a company-wide learning process (quality improvement) [ISO00].
Common standards for quality management describing how these areas can be covered are ISO 9001 [ISO00] and, with a focus on process management, CMMI [SEI06].
The following approaches represent specific strategies that may be regarded as the result of a thoroughly performed quality management.

[3] See development paradigms above.

Risk management Risk management is an approach to systematically identify and evaluate potential problems in a project early, i.e. long before the problem actually eventuates [HHMS04, p. 143]. The intention of doing this is to increase control over project success, by minimizing or eliminating potential threats. Mechanisms proposed by ISO 31000 are risk identification, risk analysis, risk evaluation, planning of countermeasures, as well as risk monitoring and introducing a risk communication system [ISO09].

Change management The organization of change is described in part 3 (Service Transition) of the IT Infrastructure Library (ITIL) [LMT07]. This part is arranging a general change management process for any kind of change, regardless its size.
According to this, a change has first to be formally requested, then registered and classified. This enables tracking and planning of changes. In consequence, changes have to go through an acceptance process. If the decision is made to act to the change request, a solution has to be designed, implemented and tested. Afterwards, the changes' meta-information is stored to be available for later analyses.
In CMMI, change management is subsumed as a sub-process of configuration management [SEI06].

Configuration management According to the American National Standards Institute (ANSI), configuration management is "a management process for establishing and maintaining consistency of a product's performance, its functional and physical attributes, with its requirements, design and operational information, throughout its life." [ANS04].
Being consistent to this definition, ISO 10007:2003 identifies various activities in the configuration management process to fulfill this requirement [ISO03]: configuration management planning, configuration identification, change control (see change management), configuration status accounting, and configuration audit.
IEEE Std. 828-1998 covers the same fields, but adds subcontractor/vendor control to the list of activities, as well as the management and communication of programming interfaces [IEE98, p. 8 f.].

2.1.2. Development model reference areas (sub-models)

The term *reference area* may be confusing: why is it called that way? Well, a development model is intended to be used as a guideline. It provides an ideal description of a development process. This description is used as a *reference* for the actual real process. The six reference areas mentioned above and shown in FIGURE 2.1 are different aspects to be regarded during a development process.
An intuitive way to denote reference areas is to call them 'sub-models':

- Product (sub-)model
- Role (sub-)model

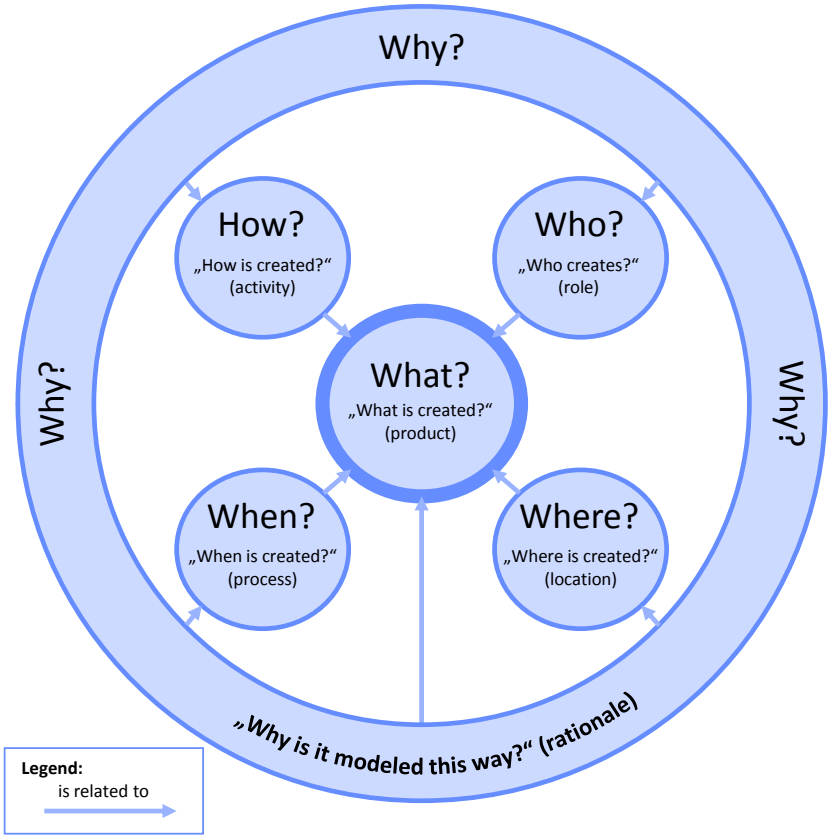

Figure 2.1: Reference areas of development models (sub-models)

- Activity (sub-)model
- Process (sub-)model
- Location (sub-)model
- Rationale (sub-)model

A product model is providing information about *what* is to be created. The role model describes *who* is creating the results. An activity model contains explanations about *how* to create, whereas a process model defines *when* these activities are performed. Activities comprise descriptions about methods and tools that are suited to support the creation of the results. The process model brings the activities into an order. The location model describes *where* a specific action is taken.[4] The rationale model is motivating the other models, explaining *why* they are modeled the way they are modeled.
FIGURE 2.1 shows these reference areas and how they relate to each other. The What is located in the center, since the result and all work products created to build this result can be regarded as the central aspects of a development model.

[4]Since software and system development is infrequently bound to a particular location of development action, the Where usually is of subordinate importance in development models. At least, it is seldom perceived as being a relevant aspect.

The V-Model XT takes a position regarding the product model as the central aspects of the development model [BMI08]. It states that it's the results that quality management is primarily about.[5] The correlated product model provides a description of the results: "What is created during the development process?".

It may as well be argued that activities are the central aspects of a development model, as it is realized in the Rational Unified Process (RUP) [Wik10]. While adopting this point of view, one would replace the How and the What in FIGURE 2.1 with each other.

Note that a development model does not exactly describe the items actually created in a development process, but the *types* of items that are to be created. For example, it may define a product type like 'system specification'.

The creation process can be supplied with additional information, provided by the other models. The process can be further specified by a role model ("Who is the creator in the creation process?"), a process model ("When is the creation process performed?"), an activity model ("How is the creation process performed?"), and a location model ("Where is the creation process performed?"). As we observed in the product model, these sub-models also describe types, not concrete instances. I.e., a development model may define role types like 'project manager', and activity types like 'testing' etc.

Finally, there is a model describing the rationale of concrete elements within the other reference areas ("Why is this element modeled the way it is modeled?"). Note that the rationale model is less of use for an actual development process, but for the development model itself, as it explains why certain design decisions were made during its development. As such, it is very useful when discussing the structure of a development model, both when analyzing its efficiency as development model, as well as when intending adaptations of the development model.

A development model does not need to cover all reference areas. All reference areas taken alone may remain unspecified and implicit, as long as there remain reference areas covered by the development model.

An important note in this context is that if a development model does not provide information about a particular reference area, then its up to the people performing a development process to make up a solution for missing areas. They can make this implicitly (producing no process documentation) or explicitly (by documenting). They can even be either aware or unaware of the fact that they must finally give a solution to the questions arisen within all reference areas.

[5] Of course, process quality etc. plays an important role, too. Yet this kind of quality is not needed for its mere existence, but finally for the quality of the product.

2.1.3. Common development models

SPEM based models

The Rationale Unified Process (RUP) is an IBM-driven development model [Kru00] based on the Software Process Engineering Metamodel (SPEM). It is used as an adaptable framework to be tailored to specific needs prior to its application.
It provides processes and process descriptions for six engineering disciplines: business modeling, requirements, analysis and design, implementation, test, and deployment [Kru00]. These are implementations of some of the approaches for successful project execution listed above, like stagewise, iterative, and incremental development.
In addition, RUP provides three supporting disciplines: project management, configuration and change management, and environment management. While project, change, and configuration management exactly map to the time and budget related approaches in development models, environment management plays an additive role as it is used to customize a RUP description to an organization or a project. This property will be regarded below in more detail, as it is a mechanism for variant creation provided by the development model itself. This is an important characteristic of an adaptable development model.
SPEM suggests various sub-models: product, activity, role, process [JBR99]. These are used in a Unified Process to fill in the different disciplines.
There are many variants of RUP [Wik10]:

- Unified Process – The generic Unified Process.
- Open Unified Process (OpenUP) – An open source variant, based on the Eclipse Process Framework (EPF).
- OpenUP/Basic – A reduced version of OpenUP.
- Agile Unified Process – A subset of RUP, integrating agile approaches.
- Essential Unified Process (EssUP) – A simplification of the Agile Unified Process.
- The Unified Process for Education (UPEDU) – A subset of RUP used for education.
- Enterprise Unified Process – An enlarged version of RUP. Includes software purchase, production operations and support, product retirement and replacement etc.

V-Model 97

The combined definitions of the term 'development model' (DEFINITIONS 1 and 2) show a certain resemblance with the way the V-Model 97 was described. In fact, the constitutional structure of the V-Model 97 can exactly be mapped to the general notion of development models in this thesis. FIGURE 2.2 is illustrating

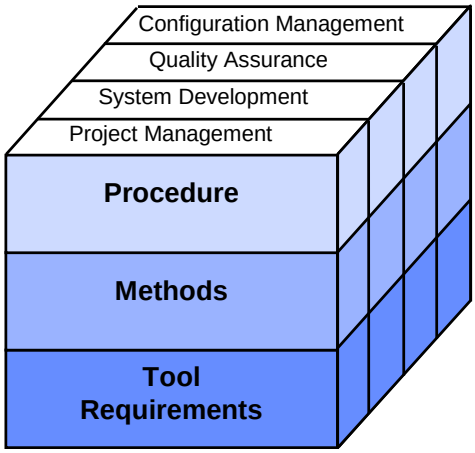

Figure 2.2: Dimensions of the V-Model 97 [BMI97, p. 4]

the cube that was used to describe the three level standardization concept of the V-Model 97 [BMI97, p. 4]. The cube is describing two orthogonal dimensions: One dimension is covering Procedure, Methods, and Tool Requirements. These are describing *what* is to be done and *how*.[6] The other dimension is divided into the sub-models project management, system development, quality management, and configuration management.
We can see that the areas covered by the V-Model 97 are a subset of the properties used to define the term 'development model'. According to the terminology chosen here, the first dimension in the V-Model 97 maps to the reference areas of a development model, whereas the second dimension maps to the approaches for successful project conduction.

V-Model XT

The V-Model XT is the development standard for IT Systems of the Federal Republic of Germany [BMI10]. Its primary objectives are:

- "Minimization of Project Risks [...]
- Improvement and Guarantee of Quality [...]
- Reduction of Total Cost over the Entire Project and System Life Cycle [...]
- Improvement of Communication between all Stakeholders [...]" [BMI08]

These objectives cover the purpose of development models as identified by DEFINITIONS 1 and 2. The solution provided by the V-Model XT to fulfill these objectives is to offer process modules containing the necessary assets needed by project

[6] From the point of view chosen in this thesis, the How covers both methods, as well as tools supporting these methods.

management, quality assurance, system development etc. [BR05]. Naturally, the preferential development strategy is the V-shaped development paradigm.
Orthogonally, the V-Model XT is organized in sub-models, providing product, role, activity, and process model.

HERMES

HERMES is a method for carrying out software development projects [ISB03]. It is an open standard provided and maintained by the Swiss Federal Administration, initially developed in 1970 [ISB09].
Its description covers three sub-models: product, activity, and role [ISB03, p. 9]. In addition, HERMES describes supportive disciplines[7] like project management, quality management, risk management, configuration management, and project marketing. Project marketing is a field of project external communication made explicit.
In 1995, HERMES' content was orientated on what was supplied by the German V-Model [ISB09].

PRINCE2

Except for the early days after its initial development in 1989, PRojects IN Controlled Environments (PRINCE) has been a generic project management method, without a focus on information systems project management [OGC09a]. Nevertheless, it is listed here for completeness, as it is fully comparable with native development models with a single exception: its purpose does not fully fit into the definition of 'development models'. PRINCE is not meant to support managing projects used "for the creation of software and systems" (see DEFINITION 1), but more generally for any development projects.
The current version PRINCE2:2009 is the *de facto* standard method used for project management in the UK [OGC09b]. It provides various sub-models: product, activity, role [OGC02]. As a generic project management standard, it extensively covers supportive areas of project conduction, covering quality, risk, change and configuration management.

2.2. Development model adaptation

There are two kinds of adaptations applicable to development model that are in the focus of this thesis: organization-specific and project-specific adaptations.
Organization-specific adaptations represent the main focus of this thesis. A fitting term for the adaptation process is *customization*. The proposed approach for

[7] In the HERMES terminology, these disciplines are denoted as 'sub-models'.

the adaptation of development models is intended to support the creation and maintenance of organization-specific variants of development models.
The other aspect of development model adaptation is regarding a particular project's needs. This is often referred to as *tailoring* and implies the reduction and composition of a development model to best fit to a project's application environment.
In this thesis, both aspects are integrated into the solution concept.

2.2.1. Conformity

In case of the V-Model XT, model conformity of organization-specific customizations to the standard plays an important role. The V-Model XT is mandatory for all development IT projects of the public sector. Thus, any organization developing systems for German administration may have to account for the conformity to the V-Model process.
As a consequence, a conformity program has been founded that includes personal and organizational certifications for the V-Model XT. It includes an information approach that can be used to verify the conformity of a particular customization. In addition, under certain preconditions, a variant that is using the variant mechanisms embedded in the V-Model XT since its version 1.3, may automatically be evaluated as a conform model.

2.2.2. Timeline of model evolution

As FIGURE 2.3 illustrates along a cutout of the version and variant history of the V-Model XT, the adaptation of development models is characterized by two dimensions: further development and the creation of variants. A variant can itself be subject to further development and variant creation.
When regarding the creation of variants of development models in practice, we can observe three steps of building variants:

1. Initial development of the original development model, both meta-model and model contents. The developer is denoted here by *core asset engineer*.
2. Customization of the original model to particular needs of an adapting organization that is intending to apply the development model. The role performing this customization is the *organization engineer*.
3. Tailoring of a development model for a particular project. This step is not illustrated in FIGURE 2.3. The actor performing the tailoring of a development model is the project *manager*. The target audience of the tailored model is the *user*.

Both original model and customized model are subject to further development. The development of the original model is done by the model owner, whereas the

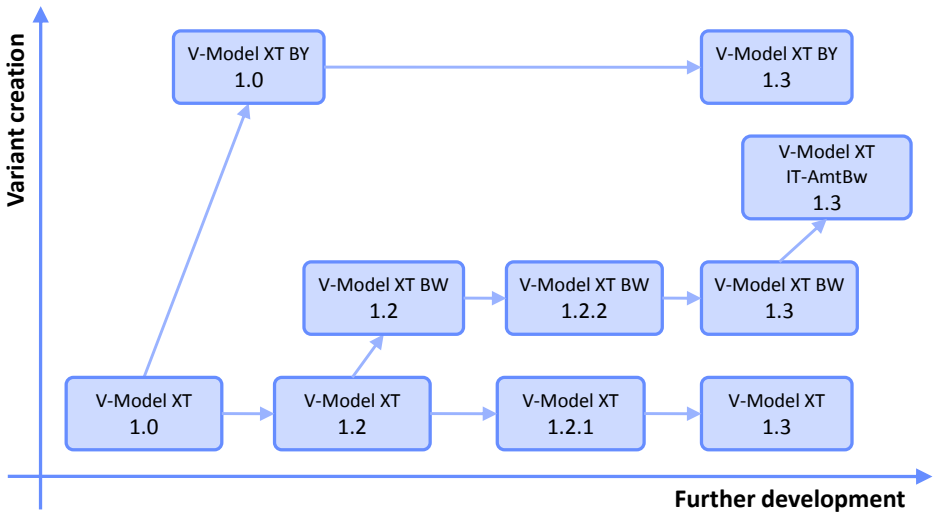

Figure 2.3: Dimensions of the adaptation of the V-Model XT

customization is developed further by the adapting organization. In practice, the development lines are often performed in parallel.
As mentioned above, the timeline of model evolution can be described by two dimensions.
The first dimension includes the decoupling of variants, as we can see in FIGURE 2.3. There have been two variants of the V-Model XT created at different times, namely the V-Model XT BY adapted by the Bavarian Department of the Interior, and the V-Model XT 1.2 BW for the German Bundeswehr. For the V-Model XT BW 1.3, there has even been a further variant, which is department-specific: the V-Model XT IT-AmtBw 1.3.
The second dimension refers to further development of existing variants. In FIGURE 2.3, there is a cutout shown from the version history of the considered models. We can see that all model variants reached version 1.3.
Not too obvious in the figure is the fact that variant creation is only one direction of the vertical dimension. Merging model branches like the VMXT BW and VMXT BY is a way to decrease differences between further developed model variants. I.e., the model were synchronized with the original development line. This synchronization was a major task for both variants.

2.3. The need for development model adaptation

Standard development models are created with a general requirement to be highly generic, in order to be applicable in a variety of project context throughout a variety of different organizations.
Specific needs and characteristics of the applying organizations cannot all be satisfied by a generic standard. ROMBACH identified lack of guidance in development

models that are too generic [Rom05, p. 87].[8]
On the other hand, when too specific, a standard process model is restricted to a subset of organizations with very specific needs.
In particular, the following problems motivate the creation of organization-specific variants of a standard development model:

- Nomenclature does not fit.
- Development processes do not fit, i.e. applicable development paradigms and organization of managements disciplines may differ vastly.
- Technologies and tools in use in an organization may not be supported by the processes defined in a development model.

There are two general ways to overcome these problems: 1. to create an organization-specific variant of a development model, and 2. to adapt the organization to the processes defined in a development model.
Since a development model is very generic and does only provide vague guidelines regarding technologies and methods to be used during development, it is usually perceived as the more promising approach to adapt the development model to the organization and enrich it with technology and method specifics.

2.4. Adaptable development models

There exist concepts in meta-model based development models that are intended to support the adaptability of the models to facilitate variant creation. These concepts are analyzed in CHAPTER 3. They were integrated in SPEM and the V-Model XT.
But the existent implementations of these concepts only provide selective solutions to the problems arising with variant creation. It is intended to provide a more holistic solution in this thesis.
Approaches to implement adaptability into development models are:

- Variability operations (SPEM) [OMG08b].
- Change operations (V-Model XT) [Kuh08, Ter09].
- Tailoring (V-Model XT) [KN05, Kuh05, Gna07, KT09].
- Modularity (SPEM, V-Model XT) [OMG08b, Kuh08].

The research group around MÜNCH, ARMBRUST and OCAMPO follow a distinctive approach to support the creation of development model variants. Their focus lies on the evolution process, not the structural constituents of a development model. This evolutionary approach covers:

[8] Note that ROMBACH compared organization-specific development models with the actual usage in a project, but this ambivalence is transitive and even more severe for a generic standard model.

- scoping development models to identify the locations where variability is needed [AKM+08, AKM+09],
- rationale support during development model evolution [OS07, OMR09, OM09], and
- differential analysis of evolved model variants [Arm08].

The first point is centered around the process to determine the properties an actual model has, and the identification of commonalities to infer needed variabilities. This approach lays its focus on the analysis of an existing model and the possibilities to create a variable derivative which can be used later on to flexibly create variants. The latter two points are both mainly focused on an *a posteriori* observation of the evolved subject. I.e. they do not explicitly support the variability of a development model, but the handling with evolving models as a whole.

2.5. Problems in current adaptations

The problems that are of significance in this thesis arise with development models that undergo a customization. In the state of the art of development model customization it is assumed that a development model is adapted to specific needs by either changing the original sources itself and thereby creating a newer version by copying and modifying the model. These changes in the standard model lead to some problems stated below.
This approach for change is unsatisfactory [Tai96, p. 451]. Direct modification is likely to cause inconsistencies with other components referring to the modified component.
Furthermore, copy&modify is uneconomical, because usually the changes "are relatively small in proportion to the overall size of the component" [Tai96, p. 451]. This leads to large amounts of copied passages or model contents that remain untouched during modification. These parts have an influence on maintenance.
Finally, the relationship between the original and the copied components is lost, causing maintenance problems with a high probability later on.
The resulting problems can be divided into two groups by their granularity. The first group of problems concerns model variant organization and conformity of variants (macroscopic). The other group contains problems that arise when editing a development model in detail (microscopic).

2.5.1. Macroscopic problems

Problem 1 ***(Updates of the reference model)*** *The reference model changes after customization.*

When the reference model, i.e. the original model that was subject to an adaptation, is updated to a newer version, it can be a major task to update an organization-specific adaptation to this version [Ter09, p. 173].
In a traditional adaptation approach, there are two strategies to address this problem. The first strategy is to analyze all changes made in the reference model and then apply these changes to the customized model. The second strategy is to do it the other way around: analyzing the changes made to the customized model and then apply them to the new version of the reference model.
Both approaches have some drawbacks:

- The analysis of the changes made is not trivial (see PROBLEM 3), neither in a pure text document, nor in a structured document form like XML documents [Arm08].
- The manual application of the changes to either the reference model or the adapted model is error-prone both in respect to correctness of the result as well as to completeness of the issues tackled with.

Problem 2 *(**Conformity of an organization-specific development model**) Under certain circumstances, when adapting a development model, it may be important that the result sticks to a given set of rules, i.e. there may be the need to observe conformity guidelines.*

For example, the German V-Model XT [BMI10] is intended to be used as a development model in development projects for public authorities. In this case, the supplier has to stick to a standardized V-Model interface to enable collaboration between acquirer and supplier.
When a supplier adapts such a development model to his organization's needs, he may be obliged to keep regard on the interface to the acquirer. So for conformity reasons, the customizing development model engineer is restricted in the degree of possible adaptations.
Such restrictions are usually provided informally, if provided at all. It is difficult to fulfill these requirements, since they are vague and the development model engineer has no guidance during adaptation.

2.5.2. Microscopic problems

It's change that is needed to create variants of developments models. In order to be able to differentiate between an increasing amount of variants, and to keep track of all variants, it is important to control the changes and the motivation of change.

Problem 3 *(**Localization of changes**) In the first instance, changed model elements cannot be distinguished from other model elements. The information about the change itself must usually be retrieved using a specific method, either by writing a separate change protocol or by differential analysis.*

Changes in adapted models are not an explicit part of the model. Instead, the model only contains an incarnation of the executed change. This makes it difficult to locate the change information itself, when this is of interest.
Lack of this change information hampers synchronization of several model variants developing in parallel. In severe cases, changes that ought to be part in all branches may be implemented incompletely or incorrectly in one or more variants.
ARMBRUST describes difficulties when finding differences between several versions of a development model [Arm08, p. 7]. A proposed solution concerns technical aspects like a differential analysis. Evolyzer [Arm08, p. 8 ff] is a tool to partially circumvent these problems for structured documents like XML documents.
But a differential approach is a symptomatic approach to PROBLEM 3, as it only provides an *a posteriori* solution to the consequences of undocumented changes. It does not cover the change itself, just its consequences.

Problem 4 *(**Loss of the rationale of changes**) Along with the changes made in an adapted model, it is important to know the rationale of the change. This information is needed to be able to evaluate changes and set them in relation to other changes that may be relevant in the contextual environment of the change.*

In a traditional modification setting, the rationale of the change is easily lost due to its detachment from the change subject. This includes loss of motivation, pro&contra, discussion, and the decision process concerning a change [OM09, p. 87]. This may result in inconsistencies or ambiguity, as well as lacking knowledge capturing and support for decision making [OM09, p. 85].
Even if the rationale is documented and stored for later usage, it is difficult to relate it to changes actually made. This is due to the observation conveyed from PROBLEM 3 that changes are not explicit in a model, but only present as their resulting incarnations.

3. On the notion of variability

Contents

Many researchers and practitioners introduce variabilities into their concepts and products with the goal to incorporate flexibility. But what exactly is variability? It is obvious that in differing contexts, the meaning of 'variability' is recognized differently. In this chapter, we will take a look at three different types of variability that emerge from different application contexts. In some contexts, we can observe several of these types all at once, overlapping.

3.1. Three types of variability

Any informational item may be subject to variability: software, systems, models, etc. The subject underlying variability is denoted here by *variability subject* .
When the term *variability* is used in the literature, we can observe different notions. In some cases, variability is understood as a property to either induce change or to identify differences between similar objects [AEH+08, AG01, AKM+08, BFG+02, DSB09, Kuh08, LW94, OMG08b, Sch06, RK08, Tai96, vGBS01, WZ88, . . .].

In most of these references, variability is additionally used as an instrument to simply extend a variability subject with additional properties. SIMIDIEVA *et al.* provide a process variation approach, which contains a concept for *pure* addition [SCO07], without any implications to changing existing properties. Interestingly, this concept is intended to provide a change in a process as a whole, by purely adding subprocesses at insertion points.
For modular systems and models, as well as for product line approaches, variability is implemented by configuration mechanisms for the assembly of a greater whole from smaller parts [AMS06, BPS04, CKK06, dOGHM05, DSB09, Kna04, LK04, PvdLM06, SvGB02, ...]. These parts can be reused and combined in other ways to create variation.
Each of these *variability types* is characterized best by the intended goal that is implied by the variability. The identified goals are:

- *configurability*,
- *extensibility*, and
- *modifiability*.

The variability mechanisms differ for these variability types. In this chapter, we take a look at properties and instances of each of these variability types, separately. A look into the notions of these terms is the way to achieve GOAL 1.
For the definitions following in this chapter, all variability types distinguish two points in time. The first designates the state of the variability subject before execution resp. interpretation of the variabilities (*initial state*). The second is determined by the state of the variability subject after variability execution resp. interpretation (*varied state*).
Note that variability can actually be executed, resulting in a changed variability subject. Or, it can be interpreted without actually changing the variability subject. Both ways for realization of variability should be transparent to the client using the variability subject in its varied state, though.

3.2. Configurability

When using the term *variability*, many researchers imply the ability of a subject to be assembled from smaller parts in various combinations. Each combination as a whole has different properties, functions, data etc. and is considered a variant.
Those small parts may be both functionally dependent and independent from each other. For the assembly, a *configuration framework* is needed, which is allowing for the definition of concrete compositions.

Definition 3 *(**Configurability**) Configurability is the ability of a variability subject to be put together from smaller assets (configuration units), i.e. assembled by choice of various options.*

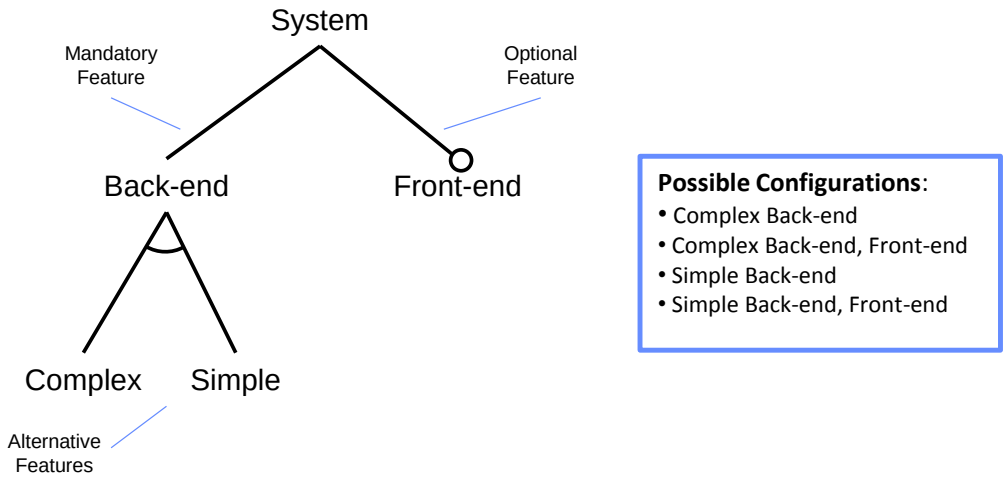

Figure 3.1: Principle of configurability

When following this definition, a configuration framework must allow the declaration of *configuration unit*s to be subject for a configuration that defines a whole. A framework has to handle the rules of composition and to provide a mechanism for configuration of a whole.
FIGURE 3.1 illustrates the principle of configurability using a simple feature modeling technique [KCH+90]. Feature modeling is a common technique to describe the structure of a software product line [CN02]. It is possible to create various variants from pre-existing assets.
In the figure, an exemplary family of systems is described. Each member of this family, i.e. each system that can be created, must have a back-end and may optionally have a front-end. The back-end may either be simple or complex. With these assets given, there are four possible configurations.
We will now take a look at several examples of existing configuration frameworks in SECTION 3.2.1. These examples will then be taken up again in SECTION 3.2.2, where we will see the general properties that make a configuration framework. It will be demonstrated how the examples realize these properties.
SECTION 3.2.3 gives some common mechanisms used in the examples to realize configurability, whereas SECTION 3.2.4 is listing some precautions that have to be made before realization of configurability.

3.2.1. Instances of configurability

When taking a look around, we can see many examples where systems or models are designed to enable variability by providing a configuration mechanism. We will take a look at three examples: software product lines [CN02], component frameworks [Sun09, Spr09], and development models.
As we will see in the examples, the exemplary variability subjects are composed

of configuration units. Configuration units can be selected or deselected, and their selection may be dependent or independent from the selection state of other units. The complete set of selected units constitute to a greater whole. The size of configuration units defines the granularity of the variability mechanism. A configuration unit may be part of another configuration unit, thus contributing to its internal structure.

Software product lines

Software product lines are used to exploit commonalities of partially distinct software products in order to achieve substantial production economies.
The following definition refers to the definition made by CLEMENTS and NORTHROP [CN02, p. 5].[1]

> **Software product line** A set of software-intensive systems sharing a common, managed set of features. All systems within this set are developed from a common set of core assets in a prescribed way.
>
> **Software product line architecture** Concentrates on commonalities in the implementation of a line of products and provides a structure that makes the derivation of software products from software assets possible [JS02, p. 3]. For the creation of a product, a set of applicable components from the base of common assets is chosen for integration. The components are tailored as necessary through preplanned variation mechanisms,[2] and integrated with any new components that may be necessary according to the rules of the product line architecture [JS02, p. 3].

The variation mechanisms are aligned towards three *feature types* [AG01, p. 110]. Feature types are used to determine whether a feature belongs to a particular product in a product line. TABLE 3.1 shows the feature types *mandatory*, *optional* and *alternative*. Features may be independent, inclusive or mutually exclusive. Variabilities can be applied at different times, i.e. at compile-time, link-time, runtime and post-runtime.
The product line architecture combines these feature types to provide fixed commonalities and controlled variabilities [Rom05, p. 85]. *Ad hoc* variabilities unique to a single product have to be developed from scratch and must fit in the architecture. They should use well designed interfaces to prevent architecture erosion [Rom05, p. 85].
Usually, in order to assess commonalities and variabilities, a commonality analysis has to be performed [OBM05, p. 273].

[1] These definitions have not been included in the List of Definitions in this thesis as they are not directly related to the proposed concept.
[2] Mechanisms to do so are parameterization or inheritance, for example.

Feature Type	Meaning
Mandatory	The feature must always be included.
Optional	The feature is an independent complement that may be included or not.
Alternative	The feature replaces (an)other feature(s) when included.

Table 3.1: Feature Types (product lines) [AG01]

The composition framework in product lines is using *feature*s to distinguish various parts of the variability subject that can be combined to a whole. This kind of *feature modeling*[3] was proposed by KANG *et al.* to be used as a management mechanism for commonalities and variabilities in system families [KCH+90]. The proposal distinguished mandatory, optional, and alternative features and described the relationships between sets of features. Feature modeling as stated in [KCH+90] was created for the Feature-Oriented Domain Analysis (FODA) and is widely known and often cited.[4]

Component frameworks

A widely known technology that can be used to achieve variability through configurability is J2EE with its various web application server implementations like IBM WebSphere, Oracle Application Server and JBoss Application Server.
A related technology is Spring. The Spring framework is an application framework for any Java application [Joh02]. It supports the J2EE principles, but is not restricted to web applications.
Support for distribution and persistence management are very important and prominent features of J2EE and Spring [Mär05], but these are out of focus in this thesis. Instead, we look for mechanisms for configuration of objects that represent an application. Both J2EE and Spring use a deployment descriptor to define how *beans*[5] contained in an application are combined to create a particular application. The specifications of J2EE and Spring declare how the information in the deployment descriptors has to be interpreted. In our context, an implementation of these specifications works as a configuration framework.
Implementations of classes to be configured have to fulfill specified criteria to enable such configurability. Beans serving the same interfaces usually are intended to be exchangeable, thus opening variation space.

[3] An example was shown in FIGURE 3.1.
[4] CiteseerX [PSU09a] recognized approx. 400 citations of this technical report on 19th June 2009. The exact search address is documented in the bibliography in [PSU09b].
[5] Beans are the configuration units.

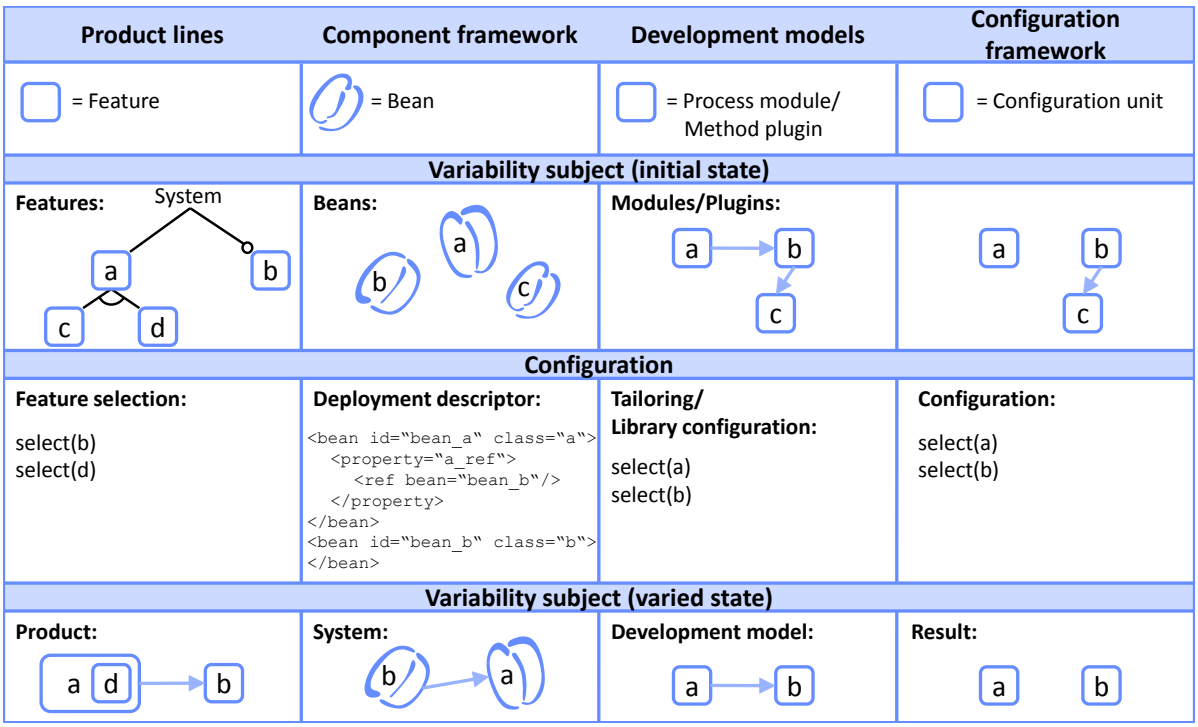

Figure 3.2: Configuration frameworks

Development models

The V-Model XT [BMI10] has a tailoring model using project types and project type variants for the project specific configuration of the development model contents [KN05]. The tailorable configuration units are so called *process modules*. Each process module is a collection of process contents that functionally and technically belong together.

Dependent on the project environment, process modules are connected with each other in predefined combinational possibilities. The resulting process module configuration is used as development model for the tailored project.

GNATZ stated that the one-to-one allocation of model contents to containers reduces the flexibility of the tailoring mechanism [Gna07, p. 169 ff.]. He proposes the introduction of structuring elements allowing one-to-many relations so that a finer granularity for the selection process is possible.

A mechanism comparable to process modules is present in the Software Process Engineering Meta-Model (SPEM) [OMG08b]. This meta-model for development models provides *method plugins*, which are containers for all content and process elements. A *library configuration* classifier adds capabilities for packaging of method plugins to enable development model tailoring.

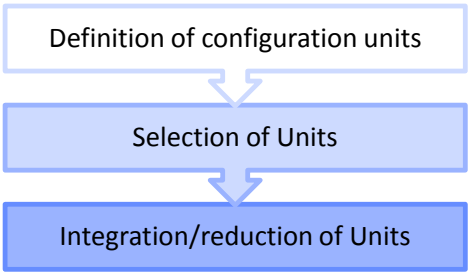

Figure 3.3: Characteristics of configurability

3.2.2. Properties of a configuration framework

FIGURE 3.2 shows the chosen examples in comparison with the notion of configurability stated above. The rightmost column is showing the generic structure, from top to bottom: a variability subject in its initial state consisting of configuration units, a configuration process, and a variability subject in its varied state. These characteristics are summarized in FIGURE 3.3.
We can see that product lines, component frameworks, and the chosen development models are implementations of a configuration framework. According to FIGURE 3.2, it is possible to identify configuration units in every example. In the chosen examples, the configuration units are features, beans, process modules, and method plugins. In addition to the configuration units, all concepts provide a mechanism for the selection of these units, in addition to an implementation for automated integration of all selected configuration units. The result is either a product, a system, or a development model.
Note that the definition of relations between the configuration units is arranged differently in the examples. In product lines and the chosen development models, the relations between features resp. process contents are part of the variability subject. For a component framework, the configuration of the relationships between configuration units is additionally dependent on interconnection information as a part of the deployment descriptor. Thus, the latest point in time to define these relations is during configuration of units.
If you can, in any variability subject, identify entities that are likely to be used as configuration units, and this subject provides a configuration mechanism with a downstream integration mechanism, then you face another example for configurability.

3.2.3. Configuration mechanisms

As the previous section pointed out, and according to FIGURE 3.3, configurability is enabled by the definition of configuration units, and a selection and integration mechanism. Configuration is then characterized by selection and integration of configuration units.

Selection. The selection step is dependent on the feature model approach chosen. In product lines, such an approach enables selection by providing notions of mandatory, optional, and alternative features. In component frameworks and development models, selection of configuration units is explicitly realized by pointing out elements to be included in the result. In general, there are no further restrictions on the selection, but we know from the V-Model XT that the selection mechanism can be enhanced by a tailoring framework providing an additional layer of selection criteria to facilitate process module selection [BR05, Gna07].

Integration/reduction. When configuration units have been selected, integration is necessary. Integration is dependent on the domain technology used for the definition of the variability subject. I.e., for software the integration rules differ from those of development models. Once connection issues are solved, integration is a straightforward domain-specific process.
Integration underlies different general conditions for the configuration of product lines and component frameworks, i.e. software, in comparison to development models.
Integration of development models after the selection process is characterized by removal of deselected items, i.e. reduction.
For software, explicit definition of variation points is necessary for the realization of integration in product lines [RK08]. Variation points define an interface between a configuration unit and the embedding configuration framework. A configuration unit must realize such an interface to be able to be integrated within a configuration framework.
Known mechanism to realize this kind of variability in process families are encapsulation, parameterization, extension points and inheritance [Sch06, SP06]. We can observe usage of these (software related) mechanisms in J2EE and Spring. The realization of these mechanisms using constructs from programming languages is done with polymorphism, interfaces, dynamic class loading and static libraries, configuration files, and conditional compilation [SP06].[6]

3.2.4. Precautions for the design of configurability

When coping with configurability as defined above, the designer should be aware of the following circumstances that may arise:

- Configurability needs to be *manageable at all levels of abstraction* [BPS04, p. 334], so when different configuration stages[7] are involved, the variability mechanisms still need to be scalable, traceable and create consistent results

[6] The examination in [SP06] is restricted to Java language, but can easily be transferred to other object-oriented programming languages.
[7] E.g. configuration at development time, linking time, runtime etc.

[BBM05, p. 184]. I.e., duplicate or missing functionality and inconsistent data structures due to chosen or removed configuration units must be avoided. This prerequisite induces the need for a mechanism in a configuration framework to guarantee integrity of the generated results. Additionally, the selection mechanism must allow the user to understand the implications of his decisions when configuring the units.

- Variability mechanisms must be designed to *handle feature interaction* [SvGB02, p. 707]. A feature interaction is "a situation in which system behavior (specified as some set of features) does not as a whole satisfy each of its component features individually" [Gib97, p. 46]. It is considered "impossible to give a complete specification of a system using features because the features cannot be considered independently" [SvGB02, p. 707].
  Feature interaction is a requirements specification problem [Zav93, p. 22]. This makes it difficult to arbitrarily combine features at later stages.[8] Therefore, the configuration framework must fulfill at least one of the following requirements:

  1. Restrict features so far that undesired interactions cannot occur.
  2. Define explicit interaction protocols for all features.
  3. Include the requirements specification into the configuration decisions.

3.3. Extensibility

Another popular usage of the term *variability* is to denote the utilization of extensions. Extensions need concepts to allow addition of structures or behavior to an existing model or system.
An *extension framework* is needed to be used for variability subjects that are to be extended. To enable the addition of extensions, such a framework must specify exact interfaces. A limitation of such a framework is that the extensions that can be realized are restricted by the existing framework structure. Thus, only those addition scenarios can be implemented that were envisioned by the developer [KL02, p. 1].
Extension can be made applicable at various times [SvGB02, pp. 711–712]. Extensibility at the development stage can be used to generate product variations, whereas runtime extensibility can be used to delay variability until the actual user of the variability subject takes a decision. The extension framework has to be designed regarding the binding time of extensions in a way that the variability subject is neither too flexible nor inflexible to additions [BFG+02, p. 16]. Too much flexibility promotes ambiguities, whereas lacking flexibility enforces workarounds or prohibits some kind of additions.

[8] Later than requirements specification.

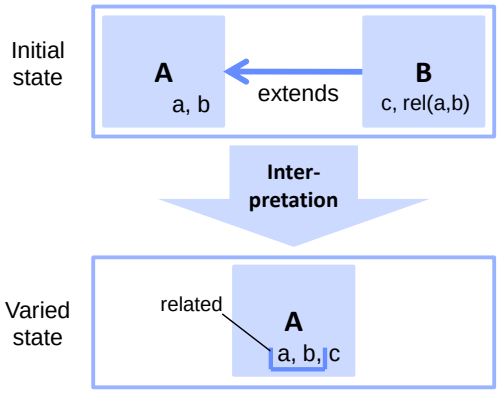

Figure 3.4: Principle of extensibility

Definition 4 ***(Extensibility)*** *Extensibility is the ability of a variability subject to be augmented by additional elements and relations between existing elements. The variability subject itself does not need to be changed for the extension, i.e. extension is realized by pure addition of information.*

FIGURE 3.4 illustrates the principle of extensibility according to this definition. An element can extend another element by referencing it. An extension framework enables the merge of both elements.

3.3.1. Instances of extensibility

We will take a look at some languages and models that were designed to be extended or to allow extension: a process definition language, object-oriented programming languages, and from the field of development models: SPEM and V-Model XT.
For software product lines, BOSCH *et al.* state that a variability framework supporting configurability can also be used for the creation of variants by mere addition of configuration units [BFG+02, p. 19], and thus realizing extensibility.

Process definition languages

Little-JIL [Wis06] is an agent coordination language that can be used for visual representation of processes. SIMIDCHIEVA *et al.* were using this language as a vehicle for representing process families [SCO07]. With this language, a process can be represented as a tree. Every leaf is regarded as atomic process step, and it is up to the assigned actor performing the step to decide what exactly is to be done during the step.
A complete process tree can be refined by pure addition of sub-nodes for leafs. This splits up the atomic step into further process steps, moving the property of atomicity to the newly added leaf. This refinement leaves all existing specifications as they are.

Object-oriented programming languages

An (object-oriented) framework is "a set of classes that embodies an abstract design for solutions to a family of related problems" [BMMB97, p. 3]. Most object-oriented frameworks provide means to implement a system by adding new classes that build upon existing functionality [Weg90, p. 36]. A thoroughly analyzed approach to do this is inheritance of abstract classes [Tai96], which is an important design technique of object-oriented frameworks [Joh91, p. 3].
A framework should be designed in a way that makes an appropriate use of the framework possible, without the need for changes directly in the framework. The definition of new classes is always done by additional, independent files with references to class names in the framework. In particular, the information stating the extends-relation is part of the extending model, not the extended one.
The compiler combines the functionality of the framework with the newly added objects. In the most simple cases, this combination does only need additions, e.g. new methods and fields. In more complex situations, inheritance can be used to overwrite methods or fields coming from the framework.[9]
Method overwriting[10] is an example for modifiability,[11] and is not covered by extensibility.[12] So when sticking to the denotations made in this thesis, we must classify object-oriented languages as an example for two principles, extensibility and modifiability.
Note: a common way to implement extensible systems is to design loosely-coupled components [BMR+96, p. 406]. This kind of extensibility relies on logical separation of concerns and narrow interfaces. In contrast, the view on object-oriented languages here is regarding extensibility of classes and their children, not components or systems. To make that clear: this thesis focuses on the notion of extensibility as defined in DEFINITION 4, not on the notion perceived in [BMR+96] and other component-oriented publications.

Development models

For extension, the V-Model XT uses a separate extension model referring to a reference model that is to be extended [BMI10]. The extension model may reference any element in the reference model, provided that the meta-model allows such references. The extension mechanism technically merges both models and generates a customized result model.
In particular, the V-Model XT has been designed in a way that many relations that often are subject to extension are first-class instances. This allows for the addition

[9] Overwriting here has the meaning of copy&overwrite, so the original functionality is not lost, but ignored.
[10] Also known as 'overriding'.
[11] See SECTION 3.4.
[12] Note that method overloading, i.e. the provision of another method with the same name, but different parameters, is a pure extension.

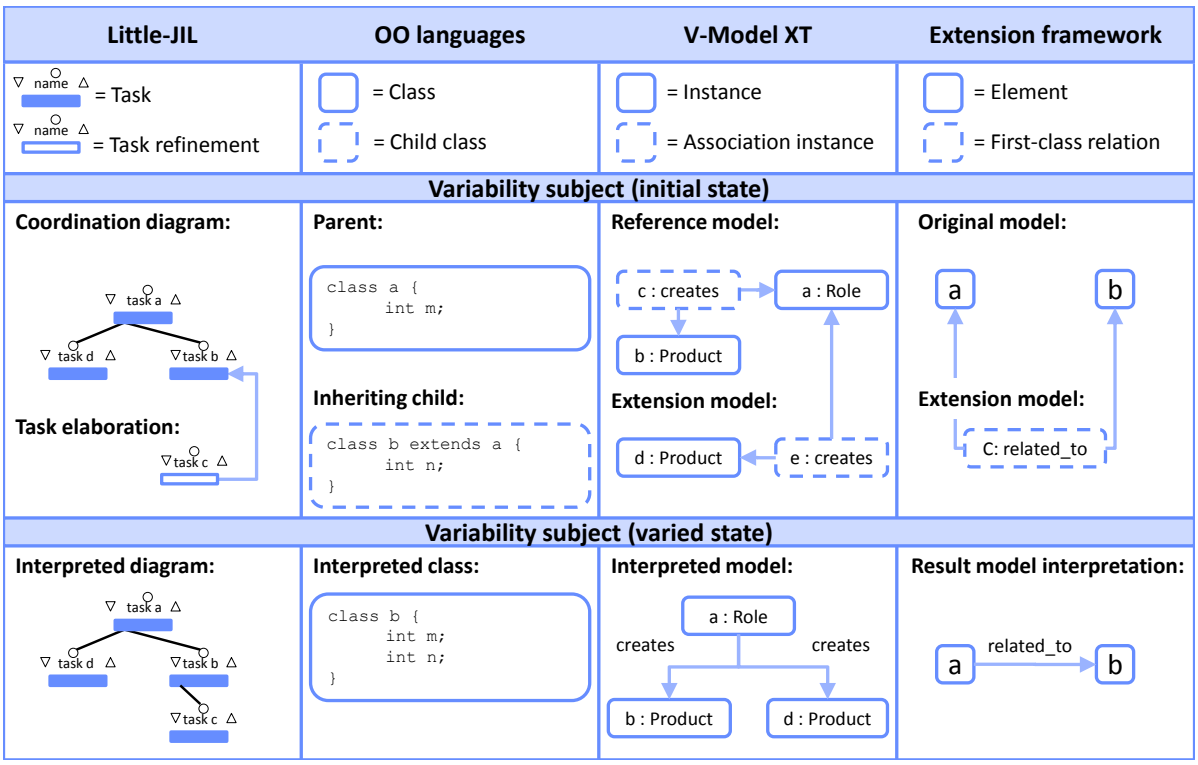

Figure 3.5: Extension frameworks

of relationships in an extension without altering the original model.

3.3.2. Properties of an extension framework

We will now take a look at the chosen examples and their relation to the notion of extensibility stated above. FIGURE 3.5 is intended to support thereby. The rightmost column is showing the generic structure of an extension framework. It relies on a variability subject in its initial state with an element that is to be extended. In the example in FIGURE 3.5, the variability subject consists of a single element 'a'.
In addition, in an extension model, an additional element is provided and an instance of a first-class relation between this element and the element to be extended. The last of the necessary properties of an extension framework is the integrated variability subject in its varied state, after interpretation of the extension relation. The illustration indicates that the related elements *a* and *b* act together as if related to each other by a second-class association. The difference in modeling the relation is transparent to the user.
Little-JIL, object-oriented languages and the chosen development models are implementations of such an extension framework.
As we can see in FIGURE 3.5, extensions can be identified in all examples. The extensions are a newly added task in Little-JIL, a newly added class in object-oriented languages and a newly added process element in the V-Model XT. The original tasks,

classes, and process elements remain untouched.
The extension itself is declared by extension relations like task elaboration (Little-JIL), declaration of inheritance, and first-class relations. In addition, all examples provide a mechanism for automatic integration of the declared extension. The result is either physically or virtually[13] created.

3.3.3. Extension mechanisms

Two kinds of extension mechanisms to be regarded when implementing extensibility in the notion used here can be identified:

- Usage of association classes
- Usage of interfaces

Association classes

A way to make any element extensible in respect to its surroundings is to refrain from extensive usage of outgoing references within the element. If an element contains all information about to whom it is related, the conclusion is that it is difficult to create additional relationships between this element and others without directly changing the element itself.
Usage of first-class associations can therefore largely enhance extensibility [Nob00, p. 73]. If an association can separately be instantiated that represents a relationship between existing elements, such a relationship can be established without changing the involved participants. Speaking in terms of UML, association classes provide this kind of extensibility.

Interfaces

Interfaces play an important role for the implementation of extensibility into a variability subject. They define the minimum criteria that have to be fulfilled by an extension unit.
SCHNIEDERS and PUHLMANN identified encapsulation as a basic extension mechanism where application-specific implementations are inserted into an invariant interface [SP06, pp. 589–590].
In development models, interfaces are determined by the type structure of its metamodel. If a work product can only be related to a role via a typed first-class relation object, this type signature represents the interface an extension has to adhere to.

[13] Object-oriented languages: an inheriting class does not physically exist, but its properties are transparently interpreted by the runtime environment.

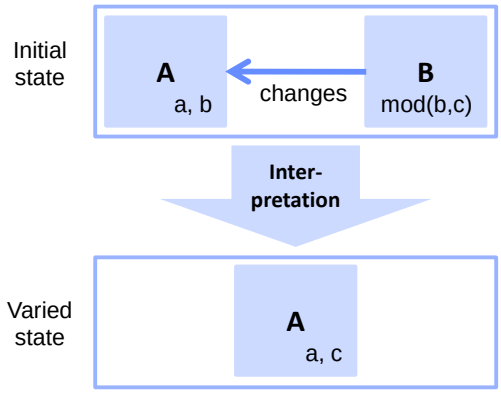

Figure 3.6: Principle of modifiability

3.4. Modifiability

The two types of variability considered so far were both concerned with existing structures that themselves remained unchanged. Variants were induced either by creative combination of existing assets (configurability) or by addition of new functions or model contents that are incorporated via first-class relations (extensibility). We will now take a look at the case when there is need to perform changes on predefined assets. This use case appears to be appropriate if a variability subject provides lots of functionality or data structures that are needed in a particular application context, but have to be modified in order to be suitable in a new environment.
In general, we can distinguish two approaches of modification:

1. Direct modification of existing code, functions, structure, data etc. I.e., if a modification is performed, the original program or model is changed.
2. Indirect modification by explicit declaration of changes to be executed by an interpreter.

Direct modification overwrites the initial state of a change subject. It is used as a common approach for further development and even for variant creation. Direct modification is one of the involved factors leading to most problems arising during adaptation (see CHAPTER 2). If only direct modification is possible, we need mechanisms to cope with the locality of changes (PROBLEM 3), as well as the rationale behind the changes (PROBLEM 4). Furthermore, we can not simply exchange the modified model with a newer version of the reference model, because exchanging the model would mean removing all modifications done (PROBLEM 1).
Indirect modification on the other hand has need for a *modification framework*. Such a framework interprets declarations of indirect modification and performs the changes in the change subject. This is a controlled transformation of the initial state into a varied state.
The principle of indirect modification is illustrated in FIGURE 3.6. An element can modify another element by referencing to it. A modification framework enables

the integration of both elements by performing an appropriate change operation. The interpretation of the changes-relation is to change properties of the element to be modified.
In the domain of software configuration management, the evolution of a change subject can be tracked either with a state-based approach, or a change-based approach [CW98, p. 262]. I.e., in the first case, the difference between two versions are derived *a posteriori* by comparing two states. In the latter case, the difference is descriptively described by explicit entities representing the change and, in consequence, the difference. Usage of change operations provide the opportunity to adopt a change-based versioning approach.

Definition 5 *(**Modifiability**) Modifiability is the ability of a variability subject to be modified by usage of specifically designed mechanisms for indirect modification.*

If a variability subject can be modified only by direct change of its properties without usage of explicit modification mechanisms, then it cannot be categorized as modifiable, according to the notion in this thesis.

3.4.1. Instances of indirect modification

In this section, we will take a look at examples that implement indirect modification as described above. The means to do this is to create modification units that reference an element that is to be modified. An interpreter is realizing this change description.

Model versioning

EMFStore [KH10a, KH10b] is a version control system for models based on the Eclipse Modeling Framework [Ecl10b]. Versioning in EMFStore is operation-based. I.e., changes performed on a model are not only executed within the Eclipse editor, but a change protocol is recorded along with each modification [HK10b]. A multitude of different change operation types are implemented into EMFStore that allows the model based documentation of descriptive changes [HK10b, Fig. 2].
Change operations distinguish between primitive and composite operations. Composite operations represent a group of primitive operations that represent a semantic unit with enlarged change semantics.
The versioning concept in EMFStore relies on such change operations. When a model is synchronized with the model repository after the modification, only the model based change information is transfered to the server.
An implementation of a development environment for models that is building upon EMFStore is UNICASE [HK10a].

Data modeling languages

XMI is a standard provided by the Object Management Group (OMG) for exchanging metadata information via XML [OMG07a]. It integrates MOF [OMG06] and UML in an XML based language.
As interchange format, XMI provides a mechanism to reduce the amount of data to be transfered when a model is changed remotely. Instead of transmitting a complete model, it is possible to specify only the differences. These differences have to be interpreted transparently, so the user will only act on the changed model without necessarily knowing about how the model emerged.
For this purpose, XMI defines three difference types: *Add*, *Delete*, and *Replace* [OMG07a, p. 12]. Elements of these types can be added to a model, resulting in the differences being interpreted by an XMI parser.

- Delete induces the removal of an element with all its contents.
- The Add operation adds contents to a referenced element.
- Replace removes an element from the model, with it being substituted by another element. The contents of the removed element remain. They are appended to the contents of the added element. I.e., only the referenced node is replaced. All contained nodes remain.

These mechanisms allow for the descriptive definition of changes. They allow the addition and removal of XML elements, but do not allow fine-grained changes of XML attributes.
It has been proposed to use this mechanism to support the description of model evolution [KR01] and to use it during migration of persistent object models [FR05].

Development models

SPEM's method plugins are not only elements used for configuration of a development model's contents as described in SECTION 3.2.1. The method plugin package also defines modification mechanisms for method content and processes [OMG08b, p. 117]. The mechanism allows the change of contents "without directly modifying them, but by describing changes from a separate unit"[14] [OMG08b, p. 117]. Method packages can be used as collection of method plugins to enhance the possibilities for configuration of a complete process. The SPEM provides a variability mechanism that is intended to realize modifiability, in the way this term is understood in this thesis.
The mechanism includes a *Variability Element* which is used for elements to reference their basis (parent) and select the variability algorithm[15] to be used.

[14] Namely, the extension unit is the method plugin.
[15] In SPEM, these algorithms are denoted as 'variability types'. This thesis treats them as 'variability algorithms' to prevent confusion with the three variability types defined in this chapter.

The possible variability algorithms are *contributes*, *replaces*, *extends* and *extends-replaces* (see [OMG08b, pp. 137–140]).

- **contributes.** When defining a 'contributes'-relation, the variability element injects its "properties into their base Variability Element without directly altering any of its existing properties" [OMG08b, p. 137].
- **replaces.** The base variability element (parent) "is logically replaced with this new variant of the element to which all incoming associations still point as before, but which has potentially new attribute values and outgoing association properties" [OMG08b, p. 138].
- **extends.** 'Extends' allows reuse of elements. "The result of interpretation is that the special element has the same properties as the based-on has, but might override the inherited properties with its own values" [OMG08b, p. 139].
- **extends-replaces.** "Extends-replaces combines the effects of extends and replace variability" [OMG08b, p. 140].

Since version 1.3, the V-Model XT has a similar mechanism for the definition of indirect modifications [Ter09, p. 174]. Its meta-model offers concrete *change operation*s that can be instantiated to define changes. KUHRMANN named change operations in the V-Model XT and distinguished them from extension operations [Kuh08, p. 139], which do not perform changes, but additions. Change operations are used for the logical manipulation of a model.
For example, the organization engineer can rename a role not by change of the name field of a particular role (direct modification), but by addition of a new instance of *RolleUmbenennen* (RenameRole). This instance must reference the role to be renamed and contains the information about the new name.
The meta-model contains change operations for all use cases of modifiability intended by the V-Model XT authors. Cases not provided or envisioned by the authors are not allowed. This effectively restricts flexibility.

3.4.2. Properties of a modification framework

Like we did for configurability and extensibility, we will now run through the instances of modifiability and compare their structures with the notion of a framework for indirect modification. FIGURE 3.7 is intended to help with this. Since direct modification has no need for a framework to perform changes, a *modification framework* is always used to realize indirect modification, and thus refers to indirect modification.
Like before, the rightmost column in the figure shows a generic structure for modification frameworks. It has a variability subject in initial state that is about to be modified. A modification unit is used to declare modification of an element via a modification relation. In the result, i.e. the variability subject in its varied state,

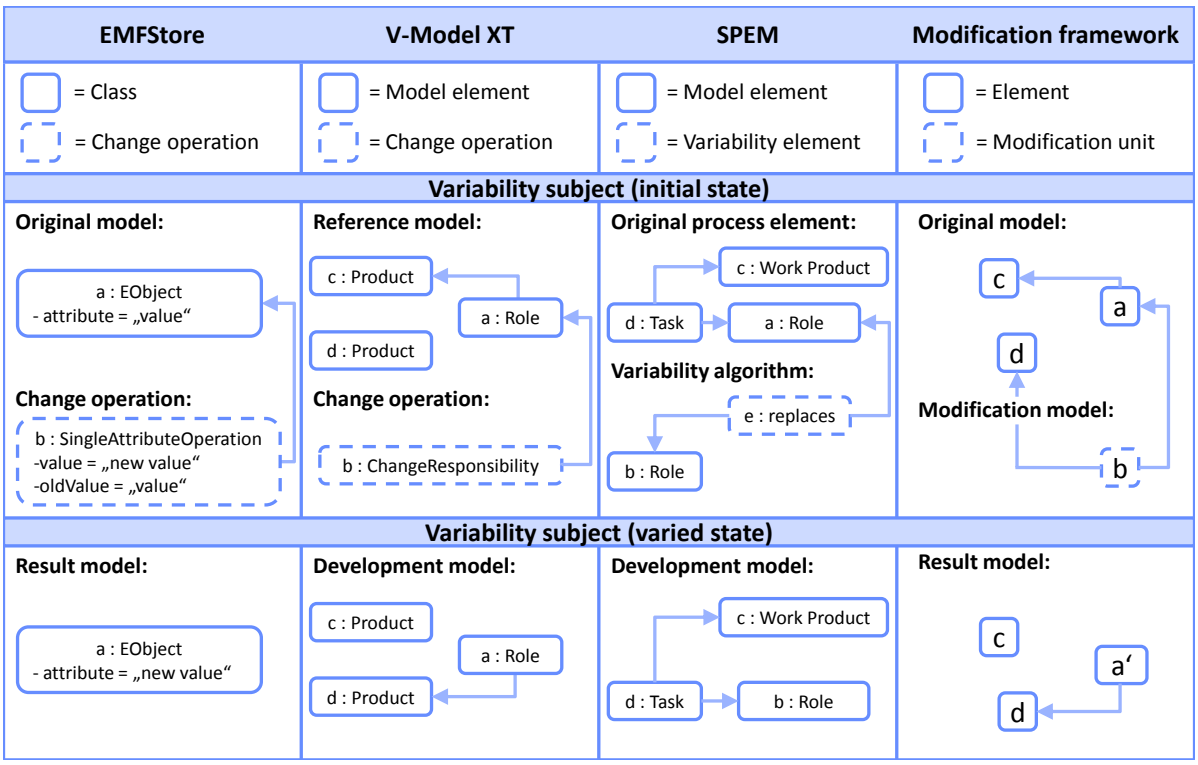

Figure 3.7: Modification frameworks

some properties of the element to be modified have changed. In the illustration, a relation from element *a* to element *c* has been replaced by a relation to element *d*. Since the element *a* is not the same as before, it is marked as *a'*. This does not necessarily mean that the name of *a* has changed.

In addition to the generic structure, FIGURE 3.7 is showing modification units in EMFStore and the development models V-Model and SPEM.

EMFStore provides mechanisms for change recording. These recordings use a change operation meta-model that is generic for all EMF models. The exemplary element *b* is a change operation of type *SingleAttributeOperation*. Such a change element is created during change of an attribute in an *EObject* like *a*. During change recording, not only the new value to be set in the target element is attached to the change operation, but the old value, too. This enables undoing changes if requested by the user.

In the V-Model example, there is a change operation *b* of type *ChangeResponsibility*. It points to a Role *a* which is responsible for Product *c*. The change operation declares that *a* is to be made responsible for another product, namely *d*.[16] In the resulting development model after interpretation of the change operation, Role *a* is not responsible any more for Product *c*, but for Product *d*.

A similar example can be seen for SPEM. In the exemplary case's initial state, Role *a*

[16]The relation to the new product must be modeled, but is not visualized in FIGURE 3.7.

is performing Task *d*. In order to declare a modification of this state, a new Role *b* is defined with an embedded variability algorithm of type 'replaces'. It points to Role *a*. After interpretation, Task *d* is performed by Role *b*. The variability operations does not only affect the referenced Role *a*, but all process elements with an outgoing reference to Role *a*, too.

3.4.3. Modification mechanisms

The way to implement indirect modification is to implement change operations into the modification framework. The framework must provide a meta-model containing elements that represent change operations able to reference to elements in the model to be changed.
In addition, the intended semantics for each operation must be specified and implemented. For example, in EMFStore, a change operation of type *SingleAttributeOperation* has a very specific semantics implemented into EMFStore [HK10b, p. 79].

4. On the notion of variant restriction

Contents

The variability mechanisms described in the previous chapter span the variant space of a variability subject. In order to prevent invalid results, it may be desired to restrict the variant space to reasonable variants. “After all, the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs” of the variability subject [SvGB02, p. 708]. FIGURE 4.1 provides an overview of the different methods that can be used to restrict the variant space. A variable model can be configured, extended, and modified. Configuration is only possible within the narrow bounds defined by a feature model. Extension and modification of an existing model is always bound to its meta-model, i.e. the meta-model is restricting the variant space of customized variants. Such a meta-model can be used as a syntax definition to check a model against. Since a syntax definition can be used to evaluate a syntax as being either valid or invalid, it spans the variant space of valid models (*syntax restriction*).
The meta-model’s ability to define a variant space can further be enhanced by usage of constraints (*constraint restriction*). Such constraints restrict the variant space provided by a syntax definition.
However, checking against a meta-model and a set of constraints may not be enough. Semantical considerations of informal information like natural language are not factored in. In order to further restrict such a semi-formal model, human interaction is needed (*manual restriction*). For example, a variant overwriting an existing description text with a contrary description may annul central aspects of a development model.
Imagine a description like “The project manager must update the project plan according to the projected course of the project”. If a development model variant instead states “The project manager initially creates a project plan and this plan is carved in stone throughout the whole project”, then we certainly had to admit that

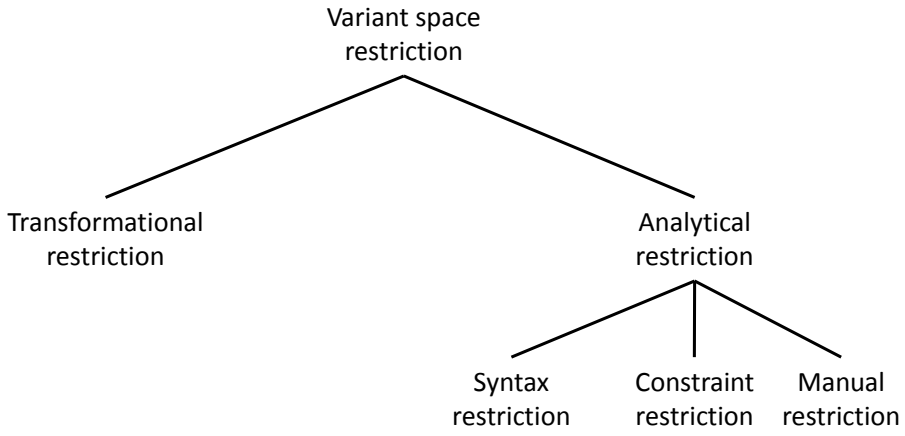

Figure 4.1: Restriction types for the variant space

the development model variant does not conform to the original intentions of the development model. Changing the text to "The project manager must update the project plan weekly" might be a valid variant of the original description. We need an analysis performed by a human to be able to achieve a fitting decision.
The mentioned restriction types, i.e. syntax, constraint, and manual restriction, represent an *analytical* approach for the restriction of a model variant space. This analytical approach has been identified by BARTELT *et al.* [BFT09, p. 4 ff.], yet lacking the breakdown into the described restriction types.
Generally spoken, analytical restriction is a rule based approach used to decide whether a variant is still a valid variant. This may be done automatically for the analysis of syntax and constraints. But it includes manually checking the conformity of variants, as well, if either the rules or the subject of the analysis are not completely formalized.
A different way to control the variant space without usage of analytical methods in the first instance is *transformational restriction* [BFT09, p. 4 ff.].[1] The transformational approach is not checking the varied state of a variability subject, but determines the generation process of the varied state [BFT09, p. 6]. In simple cases, this approach guarantees that only those manipulations can be executed that lead to a valid result.
We can find this kind of result restriction in any word processor with a user interface. Most editing steps can be identified as an example for transformational restriction. For instance, when pressing the button for 'insert table' the program should ensure that after pressing this button and maybe entering some parameters, the result is a valid document. No further checking is needed. If in a certain context, the addition of tables would lead to an invalid result, as the addition of tables in a footnote may do, then the transforming operation must not be offered to the user.
In some cases, i.e. when the model to be transformed cannot be completely formal-

[1] Note that in the paper, the authors denote this approach as 'constructive'.

ized, there may reside the need for verification of the transformation result. Then, we have to additionally choose an analytical approach.

4.1. Transformational restriction

The transformational approach (*transformational restriction*) for variant creation is realized by modification via change operations. Though, while using change operations, we are confronted by the thesis that it is generally impossible to describe the complete valid variant space transformationally only by use of a reference model and change operations [BFT09, p. 10]. I.e., applying change operations may lead to invalid results.
Therefore, when considering the use of change operations, we have to take a look at two cases. The execution of a particular change operation may...

1. ...lead to a valid result for all use cases of the operation.
2. ...lead to a valid result for none or some, but not all use cases.

In the first case, the change operation is a neat candidate to realize transformational restriction of the variant space.
However, the second case leads to the need for a decision between two options:

A) The change operation is kept out of the modification toolkit. I.e., the organization engineer may not use it while modeling modification.
B) The change operation is added to the modification toolkit, but the result may be invalid.

Option A is a clean way to restrict the result of transformational variation to a valid variant space. But it may be that the transformational variability approach is too restrictive to create the variants that need to be created. So in some cases, Option B may be the only alternative.
Option B implies that a change operation may be reasonable enough to be included into the toolkit of operations provided to the organization engineer, even if it leads to invalid results. In this case, the result has to be verified using an analytical approach.

4.2. Analytical restriction

The analytical approach (*analytical restriction*) makes use of a set of rules that is applied to the varied state[2] of the variability subject [BFT09, p. 5]. The rules are used to check whether the variability subject is conform to the intended variant space defined by the comprehension of the core asset engineer.

[2] See page 28 for the distinction between initial and varied state.

Analytical restriction can be divided into three complementary restriction types: syntax restriction, constraint restriction and manual restriction. Syntax restriction defines an initial variant space. Constraint restriction reduces the variant space that is contoured by syntax restriction by usage of constraints that allow the identification of invalid models. Syntax restriction and constraint restriction can both be completely formalized.
However, if natural language belongs to the content of a model to be restricted, these restriction types cannot be used to make pinpoint variant restrictions, since a human is needed to make the evaluations and to take a decision. For this case is manual restriction.
The following subsections describe these three types of variant restriction. In CHAPTER 8, a focus will be laid on constraint restriction. For this restriction type, a rule approach will be presented that allows the engineer of a model to specify domain-specific rules that restrain possible model variants.

4.2.1. Syntax restriction

A syntax definition in general describes rules that govern the way words can be combined to sentences. In particular, a meta-model represents a concrete set of rules resulting from an abstract syntax. This set of rules can be used to decide if a concrete syntax is valid in respect to the meta-model.
Syntax restriction is a fundamental property of language based programming and modeling. Syntax analysis is very well understood [ASU92, p. 193ff.]. As such, in this thesis, it's availability is taken as being self-evident.
The syntax rules describe a space of valid concrete syntaxes. Any variant of an existing concrete syntax must be within this variant space.

4.2.2. Constraint restriction

In order to further restrict the variant space defined by a meta-model, a constraint based restriction can be used. This restriction is based on formal rules and model structures that are checked against these rules.
Simple constraints like multiplicity ranges for object instances are usually integrated directly into a modeling language.
A common way to realize more complex constraint restriction is the usage of a constraint language [OMG10a]. For UML, the Object Constraint Language (OCL) allows the definition of invariants, as well as preconditions and postconditions. These constraints are defined for a certain context, such as a model entity. This way, constraints do not only guard static model consistency, but the environment of a particular entity, as well. OCL is side effect free, i.e. the evaluation of an OCL expression will never change a model [OMG10a, p. 5].

4.2.3. Manual restriction

When applying an analytical approach to check the validity of a modified model, in many cases there is need to have someone manually perform a mapping between the model and a set of rules. The creation of such a mapping needs human interaction, as either these rules, the involved model, or both would be written down in natural language.
A manual mapping, though, is out of the scope of this thesis.

4.3. Relation between the restriction types

The relation between analytical and transformational restriction is dependent from the particular implementation. According to BARTELT *et al.*, the approaches can mutually be interchanged with each other [BFT09, p. 6]. An analytical approach may be easier to implement and understand for simple cases, whereas the transformational approach is useful if a series of complicated operations are needed to be performed as a single step. It was stated that a variability framework must find a good ratio between both ways. In addition, it may be necessary to apply analytical methods after performing transformations.
This observation holds true when taking a look at the variant spaces opened by these restriction types. In particular, there appears to be a subset relationship between the types. Thus, when applying the different restriction types to a model domain, it appears reasonable to use a certain order of restriction application.
All restriction types operate on a given syntax. Thus, they can only be applied to a domain with a specific syntax, which is the initial point of variant consideration. A set of syntax rules intensionally describe the variant space of a domain. I.e., syntax restriction designates the maximal variant space for a given domain.
Based on one arbitrary variant out of this variant space, this thesis orientates on the creation of another variant of this variant. Naturally this variant must conform to the syntax rules, or otherwise it's out of scope of the domain. Now, the application of the other restriction types appears reasonable when we are in want of mechanisms to control the result of the variant creation. Syntax restriction can't hold off all unwanted variants.
Transformational restriction reduces the amount of variants that can be created based on a particular variant to those that can constructively be created by change operations. It effectively decreases the size of the variant space for this variant.
To further reduce the amount of variants, constraint restriction can be used to automatically discard all those variants within the variant space of transformational restriction that do not adhere to given rules.
The last restriction, namely manual restriction, acts like constraint restriction in its appearance, but human interaction is needed to decide whether a variant is still within the admissible variant space.

When taking a look at the restriction types used to finally confine the variant space, we can observe that the variant space in general can be decreased with usage of each restriction type in turn.
Note that all restriction types might impose inhomogeneous restrictions where one restriction type allows variants that are prohibited by another, and vice versa.

5. Description of the conception approach

Contents

This chapter subsumes PART II of this thesis. It draws an image of a *development model line* (DML) concept envisioned to be suitable to provide solutions to achieve the goals in CHAPTER 1 and to solve the problems stated in CHAPTER 2.
This concept bases on a *DML framework*. This framework is used to create *DML architectures*, which in turn are used to create DML's. These are the basis for the creation of project-specific development models.
SECTION 5.1 explains the different DML terms and draws an outline of the DML approach developed in this thesis. The following SECTION 5.2 explains in more detail the mechanisms used to make a DML work.
Finally, SECTION 5.3 contains a short description of the structure of PART II. This structure mirrors the strategy that has been chosen for the conception of the DML approach.

5.1. Creation of development model lines

The DML concept introduced in this section is organized in three layers: the framework layer, the architecture layer, and the model layer. This distinction was chosen to offer variability to any kind of development model, not only to a specific one. I.e., the main contribution in this thesis is a framework that allows to incorporate configurability, extensibility, modifiability, and constraint based variant restriction into a family of development models, i.e. a DML.
The framework can be used by a core asset engineer to create both a meta-model (DML architecture), as well as a model (DML) for a domain-specific development

model. Since the DML is bound to the DML framework, its general structure allows the mentioned variabilities. An organization engineer can make use of these variabilities to create an organization-specific DML that can finally be reduced to a project-specific development model.
FIGURE 5.1 illustrates the relation of the various DML-related terms defined in this section. But prior to a more detailed description of this figure, the necessary definitions are made first.
For didactic reasons, the term 'development model line' is defined below in DEFINITION 8. We begin with DEFINITION 6. Note that a description of *development model* was provided by DEFINITION 1.

Definition 6 *(**DML framework**) A framework containing types necessary for the creation of DML architectures through specialization. Types in the framework are: model element, relation (extensibility), feature model (configurability), change operation (modifiability), and constraint (variant restriction).*[1]

The purpose of a DML framework is to guarantee a certain structure for all DML architectures, which in turn guarantees that every DML created using these architectures is generally configurable, extensible, and modifiable, as well as there is a base structure for variant evaluation.
The abstract types within the framework are used as generalization for all types created within a DML architecture. *Model element* and *relation* are used to model the types in the knowledge pool of a development model. In addition, relations can be used to enable extensibility. The *feature model* is needed for configurability, while *change operations* allow modifiability. *Constraints* are used to check a specific development model against previously modeled domain-specific constraints.
The feature model is the only concrete constituent of the framework. This means that it provides a full meta-model for management of feature models, which are a structure to enable configuration of development models. All other elements of the DML framework must by specialized by a DML architecture.

Definition 7 *(**DML architecture**) Contains concrete types which are created by specialization of abstract types in the DML framework. It contains:*

- *concrete model element types and relation types to enable modeling of knowledge pools,*
- *concrete change operation types with explicit semantics, and*
- *concrete constraint types with explicit semantics.*

A DML architecture serves as a meta-model for DML's and development models. It frames the areas of interest important to a concrete DML by providing a meta-model for the knowledge pool of the development models to be created. E.g., it

[1] Four types are representatives of variability and variant restriction. The associated variability is indicated in parenthesis.

may contain a 'Role' type,[2] a 'Work product' type,[3] and a relation type representing possible relationships between both.[4]
A change operation type might be an entity 'Rename Role', and a constraint type could be 'Has to be responsible for'.
Note that extensibility in a DML is enabled by conscious usage of association classes for extensible relations during elaboration of a DML architecture. The relation type between 'Role' and 'Work product' mentioned above represents an extensible relationship, if it is realized as an association class. The DML framework offers the general type *relation* for specialization to provide the property of extensibility.

Definition 8 ***(DML)*** *A development model line (DML) contains instances of the concrete types of the DML architecture and the DML framework. Consequently, it is always bound to a specific DML architecture. A DML can be made organization-specific by more instantiation of concrete types.*
A DML can be supplied to a project manager to configure it according to a particular project, with the help of a DML environment. The result of a DML environment's engagement is a development model.
Elements of a DML are interpreted by a DML environment to realize configurability, extensibility, modifiability, and variant evaluation.

A DML contains instances within the frame given by its meta-model, the DML architecture. For example, it may contain a 'Project manager' as an instance of 'Role', a 'Project plan' as an instance of 'Work product', and an instance of a relation between both.
In addition, we might have an instance of 'Rename Role' which references the 'Project manager' instance and contains the new name 'Project leader' as additional information.
Finally, there may be an 'Has to be responsible for' constraint within the model stating that in all variants of the DML, the role 'Project manager' must always be responsible for the work product 'Project plan'.
The common entity instances represent the knowledge pool of a development model, whereas change operation instances, relation instances, constraint instances, and feature selection have to be interpreted, yet. This interpretation is done by a DML environment.

Definition 9 ***(DML environment)*** *Implements operational semantics of configurability, extensibility, modifiability, and constraints. I.e., a DML environment enables feature selection and downstream model reduction (configuration), transparently interprets first-class relations (extensibility), and interprets change operation instances (modification). Additionally, it checks the result for constraint conformity.*

[2] 'Role' may be a representative within a development model's role sub-model. See CHAPTER 2.1.2.
[3] 'Work product' may be a representative within a development model's product sub-model.
[4] Note that these are types, not instances.

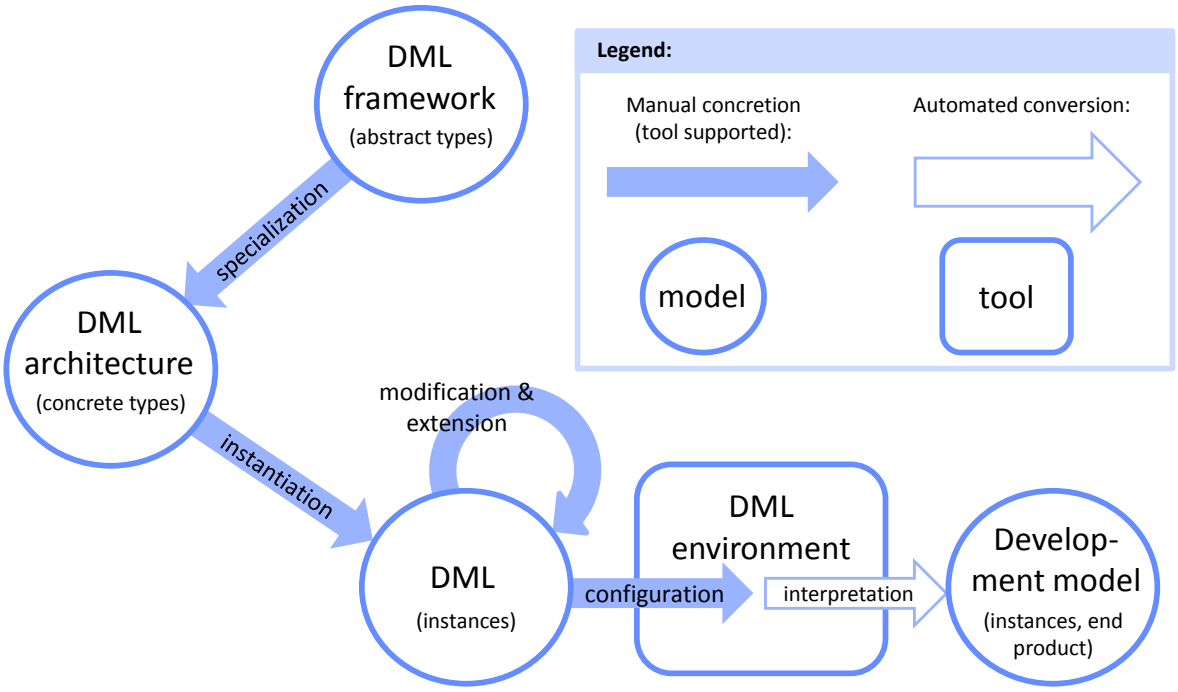

Figure 5.1: Creation of development models using the whole DML concept

FIGURE 5.1 provides an overview of the previous definitions and makes the interrelation between them more explicit. It shows that a DML framework contains abstract types to be specialized during creation of a DML architecture. Such an architecture contains concrete types to be used as a basis for the instantiation of a DML. A DML contains instances of these types.
Such a DML may optionally be extended and modified. If it is, the DML is used as a *reference DML*, i.e. it is not changed directly, but by definition of an additional model referencing the original DML. Extension and modification is declaratively realized by instantiation of elements in the *extension DML*. This is the task of an organization engineer.
A reference DML, along with any extension DML that is to be used in a particular context, is provided to a project manager to be used for the creation of a project-specific development model. The project manager selects all features applicable in the project at hand and provides this configuration setup to a DML environment. At this point, all information needed by the DML environment is available: reference and extension models, and a configuration setup. Modification elements and constraints are part of the provided models. These models can now be interpreted by the DML environment in four steps:

1. The extension DML is merged with its reference DML during a transformation performed by the DML environment. This integration may either be performed operationally, or by transparent interpretation of distinct models as a merged model.

2. The DML environment is executing a configuration of the features defined in the merged DML. To be able to do this, it is dependent on the configuration setup created by the project manager. Any model element that does not belong to any feature selected by the project manager is removed from the merged model.
3. The DML environment translates all descriptively provided modification information into actual modifications. The needed modification information is located within the model itself.
4. The resulting model is checked against constraints located within the model itself.

This transformation finally creates a project-specific development model. This development model is ready to be used in a project, by a DML user.
Each concretion and transformation step may lead to a variety of results, thus spanning a wide variant tree. The DML framework can be used to create different DML architectures. Each architecture can be used to create different DML's. Each DML can declaratively be extended and modified to create a variety of specific DML variants. And in the end, each DML variant can be used to configure a number of development models.
The difference between a DML and a development model is that a DML allows variabilities like extensibility, modifiability, and configurability. A development model is stripped off these variabilities, as it is the result of explicit variability decisions given as input to a DML environment.

5.2. Variation mechanisms of a development model line

The DML approach provided in this thesis supports the adaptability of development models within a predefined variant space. I.e., the concept as a whole is intended to support the creation of valid development model variants from a DML. It provides concrete methods usable for variant creation, as well as concrete concepts to ensure that the created variants are within an acceptable, model-specifically limited, scope. In summary, the following aspects are integrated into the concept:

- Support for the *creation* of variants (GOALS 1–4).
- Support for the *restriction* of variants (GOALS 3–5).

These aspects are illustrated in FIGURE 5.2. The arrows indicate variant creation, whereas the checkmark and the cross indicate whether a created variant is valid. The aspects are explained in more detail in SECTIONS 5.2.1 and 5.2.2. With this concept, the following questions shall be answered:

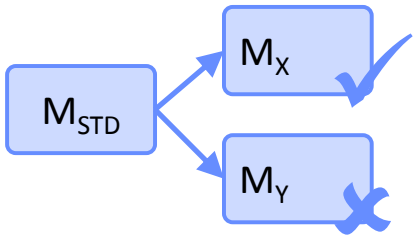

Figure 5.2: Concept properties: Create valid variants

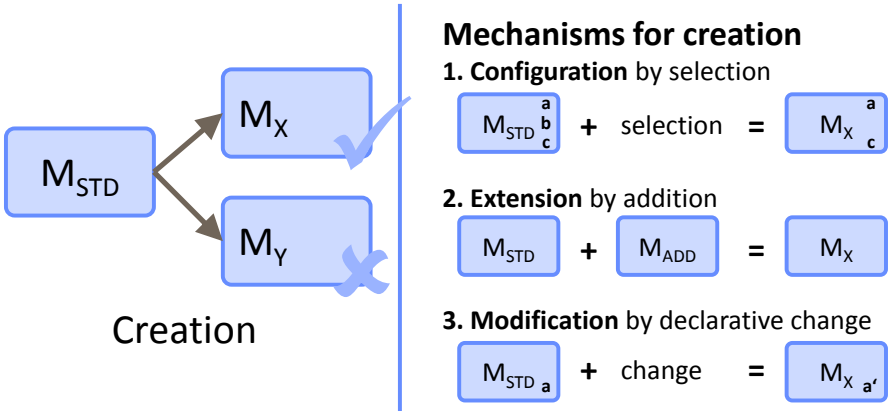

Figure 5.3: Concept properties: Create variants

How can the creation of development model variants be supported so that...

- ... we more explicitly make use of variability mechanisms like extension, modification, and configuration?
- ... we more explicitly create those variants we are *allowed* to create, not just those we *can* create?
- ... we can upgrade a variant to new versions of the original model more easily?

5.2.1. Support for the creation of variants

As FIGURE 5.3 illustrates, the creation of variants is modeled using three base mechanisms: configuration, extension, and modification.[5] These have to be provided as abstract base functionality by the DML environment.
Extension implies that variants are created by taking a base model M_{STD} and adding a differential description model M_{ADD}. M_{STD} is never changed directly. Instead, all elements contained within M_{ADD} are added to M_{STD} and merged into a result model

[5]The figure is using dark arrows and light check and cross marks to indicate its focus on creation, rather than restriction.

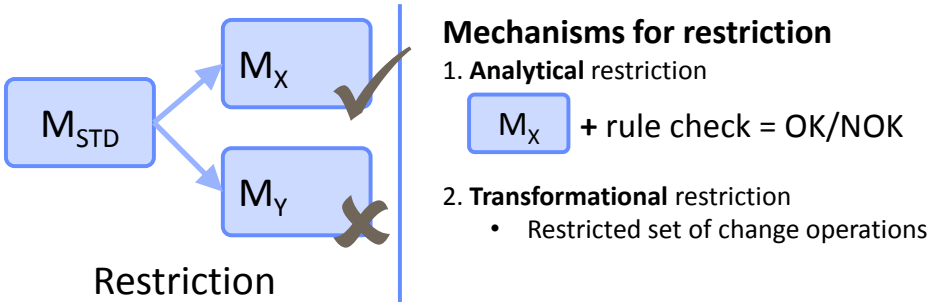

Figure 5.4: Concept properties: Restrict variants

M_X by a DML environment. This merge may be physically performed or realized by transparent interpretation of distinct models as a singular merged model.
In addition, the concept makes use of *configuration* of existing assets to tailor a DML to specific needs, generating a concrete development model.
Extension and configuration can be used to adapt development models by adding and removing contents. In order to alter contents, the concept makes use of change operations. Change operations are a declarative mechanism to realize *modifiability*: remember FIGURE 3.6 for the principle of modifiability.
A change operation, being placed itself in M_{ADD}, references another object contained within M_{STD}. The DML environment interprets the operation according to domain-specific semantics defined for this operation. The result is an altered content.

5.2.2. Support for the restriction of variants

We know that there will always be variants in the set of all possible variants that are undesirable, in respect to their intended use. So we can identify the need to reduce the amount of variants to an admissible set.
As shown in FIGURE 5.4, variant can be realized while adopting two philosophies: *analytical restriction* and *transformational restriction*.[6] Both approaches are suited to control the creation of variants. But only one of them, namely transformational variation, directly creates only those variants that are valid.
Analytical restriction is motivated when any thinkable variation can be created, since there are no restrictions set upon what may be changed. In order to assure that the created variants adhere to a set of domain-specific rules, any created variant has to be analyzed by checking it against the rules. The analysis can lead to the result that a variant is invalid in respect to the rules.
In this thesis, an analytical restriction type that is explicitly supported is constraint restriction. Syntax restriction is assumed to be present, since it is assumed that all models we discuss adhere to a concrete meta-model. Manual restriction is neglected

[6]The figure is using dark check and cross marks and light arrows to indicate its focus on restriction, rather than creation.

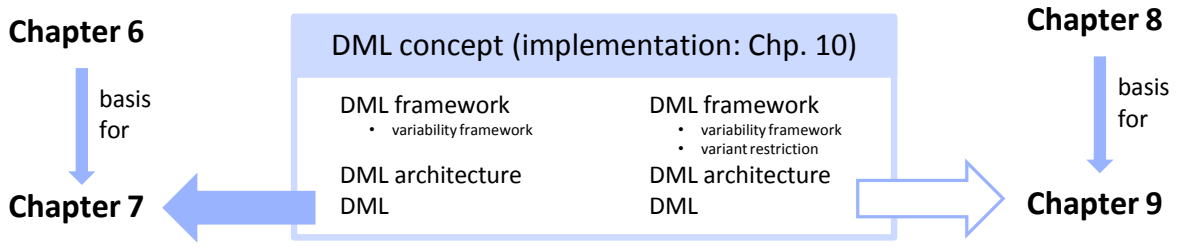

Figure 5.5: Chapters of Part II related to the proposed concept

in the thesis.
Thus, up to a certain extent, uncontrolled variation is made possible in the concept, but it will always be followed by a downstream validity analysis.
On the other hand, transformational restriction is a mechanism that only allows for the creation of valid variants, so that an *a posteriori* analysis is not necessary. This is realized by provision of a limited set of operations that apply changes to a model. These changes always generate a valid result, so that an additional check against the rule book is not necessary.
Note that this kind of controlled variation has to be implemented in a way that an operation like an 'add' operation would not be offered to the user, if the amount of elements in the model already is equal to the maximum value allowed. More precisely, the amount of elements after execution of an operation must not exceed given multiplicities. For example, a schema aware XML editor would offer an 'add' operation for elements with a multiplicity of 0..1 if and only if no such element already exists.

5.3. Concept development strategy

Part II is structured according to a strategy for the development of a variant restricted DML framework. The structure is illustrated in FIGURE 5.5. It is intended to relate the chapter structure of Part II to the proposed concept.
First of all, CHAPTER 6 (*Variability design approaches*) filters the different implementations of the identified variability types and presents design approaches to facilitate implementation of these variabilities (GOAL 2).
Afterwards, CHAPTER 7 (*Development model lines*) makes use of the design approaches elaborated in CHAPTER 6 to create a DML framework as stated in GOAL 4. This framework can be used to create DML architectures, which in turn can be used to create DML's, i.e. variable development models.
An additional dimension of variability is regarded in CHAPTER 8 (*Concept for constraint restriction*). It covers the restriction of model variants using constraints. This observation serves GOAL 3.
The DML framework presented in CHAPTER 7 is then augmented by the considerations concerning the admissible variant space made in CHAPTER 8. An enhanced

DML framework for development model variant restriction is provided in CHAPTER 9 (*Constrained development model lines*) and thus achieving GOAL 5.
Finally, CHAPTER 10 shows how the proposed DML concept can be implemented. This serves as an evaluation of the concept in respect to soundness and consistency.

Part II.

Conception of variable Development Models

Contents of the Second Part

6. Variability design approaches

Contents

This chapter is based on the notions of the three variability types defined in CHAPTER 3, namely configurability, extensibility, and modifiability.
For these variability types, design approaches are provided in this chapter to enable the creation of configurable, extensible, and modifiable systems and models.[1]
The following descriptions adhere to a pattern description template that is used by BUSCHMANN *et al.* (see [BMR+96, pp. 20–21]), with slight deviations inspired by GAMMA *et al.* [GHJV94, pp. 6–7].
Every design approach is preceded by a related design pattern. For all approaches, the following fields belong to their description:

- **Name:** The name of the approach.
- **Motivation:** A short description of the use case of the pattern.
- **Context:** A description of the situation in which the approach can be applied.
- **Problem:** The problem that is addressed by the approach, and some implications of the problem.
- **Solution:** The solution strategy used to solve the problems.
- **Structure:** A description of the structural composition of the design approach.
- **Example Resolved:** An exemplary realization of the structure.
- **Variants:** Known deviations of the approach.
- **Known Uses:** Real world examples where the approach can be identified.

[1] Note that these attributes depend on the definitions made in CHAPTER 3.

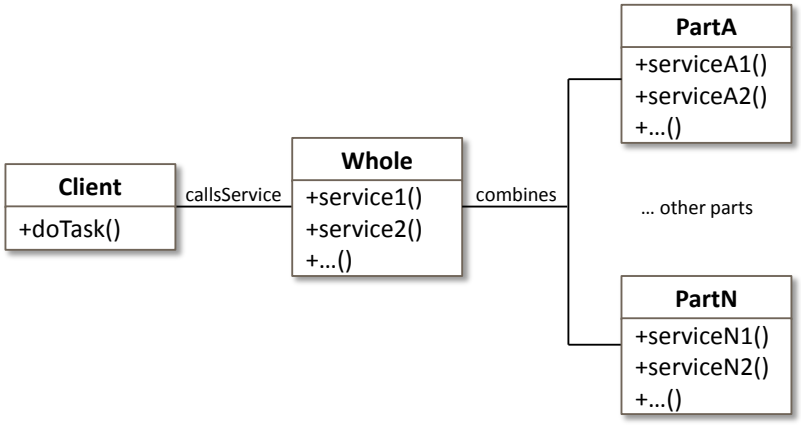

Figure 6.1: Whole-Part design pattern [BMR+96, p. 229]

6.1. Configurability design approach

This section describes a design approach for configurability that will be used in the DML framework presented in CHAPTER 7. The approach is oriented on the Whole-Part pattern.
Benefits from usage of this pattern are (see [BMR+96, p. 241]):

1. **Changeability of parts.** The internal structure of the whole can be modified without an impact on clients using the whole.
2. **Separation of concerns.** Each concern may be implemented by a separate part. This makes it possible to implement complex strategies via composition of simpler services.
3. **Reusability.** Firstly, a part's type can theoretically be reused in more than a single whole. Secondly, the encapsulation of parts provides an interface that prevents scattering the use of part objects throughout the whole.

These properties are useful for a configurable variability subject, as they help relating configuration units to each other and to the configuration framework.

6.1.1. Related design pattern: whole-part

The *Whole-Part pattern* [BMR+96, p. 225 ff.] is used to form an aggregation of components in order to subsume a semantic unit. The structure of this pattern (see FIGURE 6.1) covers three types of participants: A *Whole* object representing a group of objects, which are called *Part*s. A property of this pattern is that a *Client* can only interact with the Whole, i.e. the Whole acts as a wrapper around its constituents [BMR+96, p. 228]. Access to the Parts is only possible via the Whole. In addition, the life-span of a Part is restricted to the life-span of its Whole.
The Whole-Part pattern is related to three types of relationship [BMR+96, p. 226 f.]:

- **assembly-parts.** The Parts are tightly integrated into the Whole. Usually, this kind of relationship relies on a static structure of the Whole, i.e. the Parts don't vary.
- **container-contents.** The Parts are concentrated in a container. The contents are loosely coupled and may be dynamically added or removed.
- **collection-members.** The Parts are grouped as a collection. The collection provides functionality like iteration over all members. This is a specialization of container-contents, while all Part objects have the same type.

6.1.2. Configurability structure

Name. Configurability Design.

Motivation. In some contexts, a system or model that is uniformly used as a whole via a narrow interface may have internals that vary depending on extrinsic conditions. The constituents of the whole vary with a particular configuration made before usage of the whole. So the system's or model's composition may be configured prior to use.
In the context of development models, the project-specific configuration of a model is referred to as *tailoring*.

Context. The development of a configurable system or model: a system or model has to constitute of several units that are brought into combination by a configuration setting. The purpose is to have a flexible item that may be customized for a particular use case and be composed of predefined assets.

Problem. A design for a configurable system or model in respect to the definition of configurability (DEFINITION 3, p. 28) must provide means to...

... define a part-whole relationship between configuration units, so that all units as a whole may be treated as an aggregate object.

... select/configure a subset of units that have to behave as a part of a whole.

Solution. The solution orientates on the Whole-Part pattern (see SECTION 6.1.1). One component has to encapsulate smaller objects. Clients using this component may not access the smaller units directly. The aggregate must define an interface providing the only means to access data and the functionality of the encapsulated objects.
The three relationship types described for the Whole-Part pattern apply to the configurability approach as well.

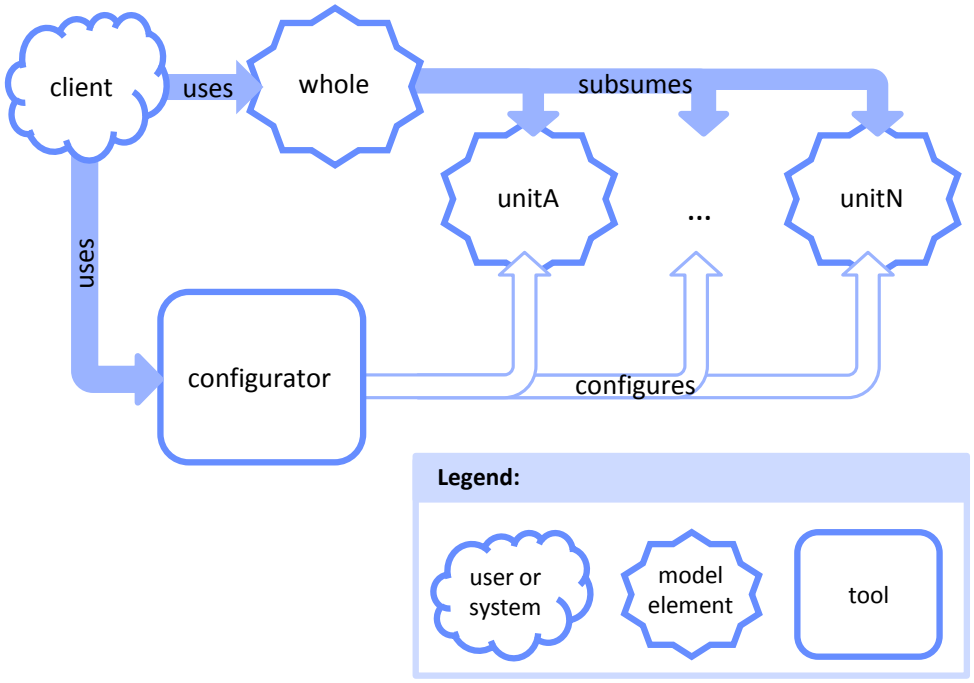

Figure 6.2: Configurability design approach

An additional aspect is the requirement for selection of smaller units to be used for the composition. This composition is done by an external configuration device enabling a client to select Units and doing the actual composition of the whole.

Structure. FIGURE 6.2 illustrates a structure that represents a realization of the solution stated above. We can see four types of participants.
A *whole* object representing a composition of smaller objects, called *unit*s. The whole is providing an interface to a *client* which relies on data and/or functionality of the whole and its units and which may either be a human user or a system. The whole encapsulates the units, direct access is not possible for the client. The only way to influence the units is either by accessing the whole, or by accessing a *configurator* component. The configurator is responsible for the selection and deselection of units and thus defines the composition of the whole. The configurator must execute the configuration prior to usage of the whole, in order to clearly define its composition.
After composition, direct access to a whole's units may be allowed for the client.

Example Resolved. It's not surprising that we already got to know an example for a configurability approach in CHAPTER 3. The Spring framework [Joh02] provides a configuration mechanism which can be used to integrate a system from smaller units, as it is illustrated in FIGURE 6.2. We will now compare Spring with the configurability structure shown in this figure to understand the connection.

```
1  class Exam {
2      Examiner examiner;
3      Student student;
4
5      void setTheExaminer(Examiner e) {
6          this.examiner = e;
7      }
8
9      void setTheStudent(Student s) {
10         this.student = s;
11     }
12
13     boolean start() {
14         return examiner.test(student);
15     }
16 }
17
18 class Examiner {
19     boolean test(Student s) {
20         return true;
21     }
22 }
23
24 class Student {
25     // burning intestines
26 }
```

Listing 6.1: Exam Java code

First, we take a look at some Java code of an exam setting. LISTING 6.1 contains three classes. There is an *Examiner* class and a *Student* class. These are the units in respect to FIGURE 6.2. The *Exam* class represents the whole. Each instance of an *Exam* must be composed of one *Examiner* instance and one *Student* instance. This is an exemplary domain-specific condition. In other contexts, one or both units used for configuration might be optional, or the same class might be used more than once in a way that would allow more than one student in an exam.
Note that the instances of *Examiner* and *Student* are not created within *Exam*. If they were, then we would not face the need for an external configurator, and thus no configurability. The methods setTheExaminer and setTheStudent are needed by Spring[2] to be able to compose the whole correctly. The start method is a domain-specific method used to begin the exam.
The classes in LISTING 6.1 provide a meta-model for an exam domain model. The Spring framework can be used to create instances of a class by defining beans. Each bean has a unique identifier and a reference to the class to be instantiated. See LISTING 6.2 for a bean configuration XML file beans.xml that is understood by Spring. This file, in combination with Spring's interpretation of the file, represents the

[2]Spring serves as the mentioned configurator, here.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3  "http://www.springframework.org/dtd/spring-beans.dtd">
4
5  <beans>
6      <bean id="ExaminerA" class="Examiner"/>
7      <bean id="ExaminerB" class="Examiner"/>
8      <bean id="StudentA" class="Student"/>
9      <bean id="StudentB" class="Student"/>
10
11     <bean id="TheExam" class="Exam">
12         <property name="TheExaminer">
13             <bean ref="ExaminerB"/>
14         </property>
15
16         <property name="TheStudent">
17             <bean ref="StudentA"/>
18         </property>
19     </bean>
20 </beans>
```

Listing 6.2: Exam Spring configuration (beans.xml)

configurator participant of the configurability approach.

In the exemplary listing, lines 6–9 define the instances of two *Examiner*s *ExaminerA* and *ExaminerB*, as well as two *Student*s *StudentA* and *StudentB*. In the shown example, the *Examiners* and the *Student* instances don't differ from each other aside from their identifier, but in an extended example, these instances would obtain additional properties like names, abilities etc.

In line 11 of LISTING 6.2, an *Exam* class is instantiated. It gets the identifier *TheExam*. *TheExam* is composed of *ExaminerB* and *StudentA* by setting the appropriate properties. The property qualifiers imply calling the methods setTheExaminer and setTheStudent on the *TheExam* instance, and providing the instances *ExaminerB* resp. *StudentA* to the methods.

With this configuration file, the composition is clearly defined. The only thing do be done now is to write a Java program that asks Spring for the *TheExam* bean. The program will get a fully initialized instance of *Exam*, according to the defined configuration. It can then call the method start to begin the exam.[3]

In this example, the Java program, which is asking Spring for the beans, is the client we know from FIGURE 6.2. The client is choosing and controlling the configuration, i.e. the configurator Spring and the configuration information in beans.xml. Thus, the client indirectly determines the composition of the whole.

[3] You may have noticed that all exams will always have a **true** as result value. This shall by no means indicate any simulation or caricature of a real exam practice. It just keeps the example short.

Variants. In some whole-part hierarchies it may be reasonable to allow a whole to contain other wholes. This case is a joint venture with the Composite pattern [GHJV94, p. 163 ff.]. Both wholes and units need to implement the same abstract interface, which is open to usage by the client. The whole is then aggregating elements implementing that interface, which can be wholes and units.

Known Uses. We learned above that Spring's configuration mechanism may be used to realize configurability.
The *V-Model XT* is another instance of the presented design of configurability. In this case, the configurability approach is translated to models. The development itself is a whole, whereas the process modules are treated as units. The tailoring mechanism of the V-Model XT is used to select process modules and thus to define the composition of the development model. With this characteristic, the tailoring mechanism represents a configurator performing the combination of units.
Note that the development model has other elements like glossary entries that are not organized in process modules, but since these elements are not subject to the tailoring mechanism, they do not fit into the declaration as a unit object.

6.2. Extensibility design approach

The premise of the following extensibility approach is the need for the ability to add information to a model without directly changing the model.

6.2.1. Related design pattern: relationship object

NOBLE presented a couple of relationship patterns, since associations are an important part of object-oriented design [Nob00, p. 73]. One relationship type he identified was *Relationship Object*, which has arisen in the object-oriented domain as a contrast to representing relationships as attributes. When using attributes, the participating objects are tightly coupled [Nob00, p 76], and a change to an attribute always involves changing the containing host.
An advantage of the Relationship Object pattern identified by NOBLE is that it may simplify the design, since relationship objects may themselves contain relationship-related information. This increases cohesion, since "other participating objects do not need to model part of the relationship" [Nob00, p. 77].
In addition, the pattern allows the separate creation of a relationship object, causing a relationship between related objects without the need to change these objects directly.
A disadvantage is the increased number of objects to be managed, since a relationship object is a first-class entity which can be uniquely identified and must uniquely be managed.

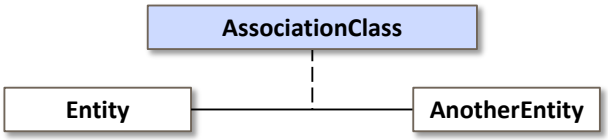

Figure 6.3: Extensibility design approach

6.2.2. Extensibility structure

Name. Extensibility Design.

Motivation. Successful reuse of already existing assets is usually recognized as an amiable strategy to prevent redundancy and to reduce development time. When building new objects, we are often in want of an archetype we can use as a basis and adapt to our needs. The extensibility design approach in this section is providing a structure that is useful when a developer is creating new structures by performing purely additive tasks to already existing objects.

Context. Development with the need for the creation of variants of already existing object. These objects must not be directly changed. A variant can thus be created by addition of information only.

Problem. A design for an extensible system or model in respect to the definition of extensibility (DEFINITION 4, p. 36) must have the following properties:

1. A newly defined element must be able to be placed additionally into an existing model context. It must be possible to neatly embed the new element into existing structures.
2. Such an element must be able to reference an element that is to be extended.
3. The original model must not be directly changed by the extension.

Solution. During meta-model design, create awareness that the usage of explicit relationship objects[4] allows extensibility. A relationship object can be added to a model to homogeneously create relationships:

- between already existing elements,
- between existing elements and new ones, and
- between completely new elements.

Structure. In UML, a relationship object is modeled using association classes. A graphical representation of such a model is shown in FIGURE 6.3. There is a class *Entity* related to a class *AnotherEntity* by an association class *AssociationClass*. This allows the creation of relationships between instances of Entity and AnotherEntity without the need to have direct access to these instances. A condition for this is that the association ends are managed within the association class, not the related classes.
The important aspect of using relationship objects as a means to realize extensibility is the design phase of a domain meta-model. If a relation between objects ought to be used to realize extension of a model, it should be designed as a relationship object. Thus, a relationship object is not only a candidate for usage if a relation is qualified by additional information, but also if it is intended to be used in an extension setting.

Example Resolved. Due to the simplicity of the approach, an example appears unnecessary.

Variants. No variants could be identified.

Known Uses. UML knows association classes to realize relationship objects. As an association class is a fully fledged class, it can store supplementary informations further characterizing the relationship, as well as operations etc.
The V-Model XT is a development model with a meta-model containing association classes that were explicitly introduced for the sake of extensibility.
Before version 1.3, the responsibility relation between roles and products was a second-class relation attached to the product. Since release of version 1.3, this relation has been replaced by an association class representing the very same relationship. This was done to allow extension models define responsibility relations to entities within the reference model, without the need for direct modification of the reference model.

6.3. Modifiability design approach

As we have seen in CHAPTER 3.4, modifiability can be achieved by either using direct or indirect modification of a variability subject. Direct modification has unwanted effects that imply most of the problems stated in CHAPTER 2. This is the reason why indirect modification is favored in this thesis. In short, direct modification might lead to...

... inconsistencies if there exist copies of the original throughout the system or model that should be modified, too.

[4] See SECTION 6.2.1.

- ... lacking knowledge at a later stage of development about what has been altered at all.
- ... difficulties when the original is developed further, and the variant is intended to be updated to the new state of the original.

The section at hand provides a design approach for modifiability using indirect modification. A common mechanism for the reuse of existing programs including indirect modification of the programs is subclassing. Inherent principles of subclassing are identified and filtered into a design approach for the realization of modifiability via indirect modification.

6.3.1. Related design pattern: subclassing

Although *Subclassing* is not a pattern in the stricter sense of [GHJV94] and [BMR+96], it may be categorized as an idiom as defined by BUSCHMANN *et al.* [BMR+96, p. 346]. Idioms are low-level patterns that provide a solution for common implementation issues. Subclassing, when viewed as an idiom, is so constitutional for object-oriented programming that it has been integrated into the literal syntax of many programming languages.
Subclassing is a synonym for inheritance and is understood as "a facility for *differential*, or *incremental* program development" [Tai96, p. 439]. WEGNER and ZDONIK formally defined inheritance as follows [WZ88, p. 55]:[5]

$$R = P \oplus \Delta R \tag{6.1}$$

In EQUATION 6.1, R denotes a newly defined object or class, P denotes the properties inherited from an existing object or class, ΔR denotes the incrementally added new properties that differentiate R from P (the delta part), and $\oplus$ denotes an operation to somehow combine ΔR with the properties of P. As a result of this combination, R will contain all the properties of P, except that the incremental modification part ΔR may introduce properties that overlap with those of P so as to redefine or defeat (cancel) certain properties of P; thus, R may not always be fully compatible with P.[6]
One might argue that subclassing is not an idiom at all, for it's an integral part of object-oriented programming languages like primitive types are. But when taking a look at languages that are not designed to provide in-language support for inheritance, like JavaScript, we can observe that a simple code archetype can be used to emulate non-strongly typed subclassing [Gol09].
This thesis is assuming that subclassing is in fact an idiom. It has consequently been integrated into the language specifications of object-oriented programming languages for better and more intuitive support of this idiom.

[5] The formalism was provided by WEGNER and ZDONIK, but we see here the notation chosen by TAIVALSAARI [Tai96].
[6] The preceding break has been copied literally from the explanation in [Tai96, p. 439].

```
1 class Line {
2     int width;
3 }
4
5 class Square extends Line {
6     int height;
7 }
```

Listing 6.3: Exemplary Java code using subclassing

```
1 class Square {
2     int width;
3     int height;
4 }
```

Listing 6.4: The interpreted result of the definition of *Square* from LISTING 6.3

We will now look at an example of subclassing in the Java language (LISTING 6.3). We can see a *Line* class with an attribute *width*. Its attribute corresponds to P in EQUATION 6.1. Another class has been added, namely *Square*. It's a subclass of Line[7] and provides an additional attribute *height*. This attribute describes ΔR.
LISTING 6.4 shows the resulting class. It's a virtual construct determining the data structures and behavior of the class when interpreted by the Java Virtual Machine. This class is the correspondence to R in EQUATION 6.1.

6.3.2. Modifiability structure

Name. Modifiability Design.

Motivation. The drawbacks of direct modification motivate usage of indirect modification. Indirect modification is modification that is descriptively modeled and performed later on by a formal mechanism. The descriptive change can be regarded as a description of the difference between the original to be modified and the intended result.

Context. Development with the need for the creation of variants of already existing objects, if these objects are likely to be updated later on in a separate development line and a merge might be necessary in future.

Problem. A modifiable system or model using indirect modification must provide means to...

[7] The subclass property is defined by the qualifier *extends* in line 5.

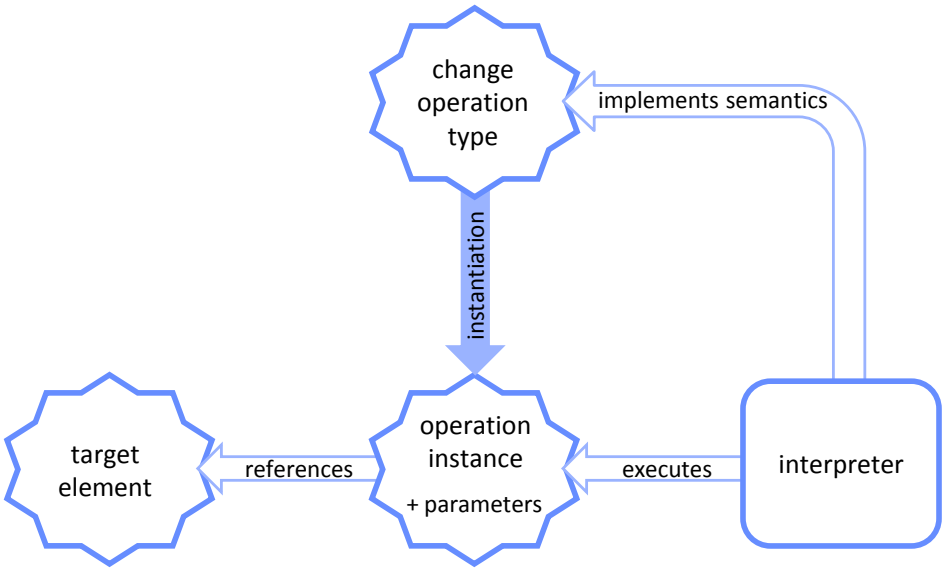

Figure 6.4: Participants in modifiability (legend: see FIGURE 6.2)

- ... model intended changes as separate change elements. This is a mechanism needed to separately declare modification without direct modification of the original.
- ... operationalize modification and to define its semantics.
- ... automatically integrate the declaration of modification with the original elements that are intended to be modified.

Solution. In order to provide indirect modifiability, the approach makes use of *change operations*. Change operations are used as a container for the declaration of modification. When modification is to be indicated, a change operation is created. An interpreter is running through all change operations and performs actions according to the semantics of the operations.

Structure. The modifiability design approach (see FIGURE 6.4) makes use of a two-leveled strategy: meta-model (upper part) and instance level (lower part).
At the upper part, domain-specific change operation types are situated for a particular change that may be performed on a model. This domain meta-model level contains the types of change operations that can be used on the instance level. For each change operation type, specific semantics must be provided and implemented in an interpreter.
At the instance level, the interpreter is then used to execute changes for every instance of a change operation type. Such instances reference target elements in a domain model and may be supplied with parameters to be used during the actual course of the change.

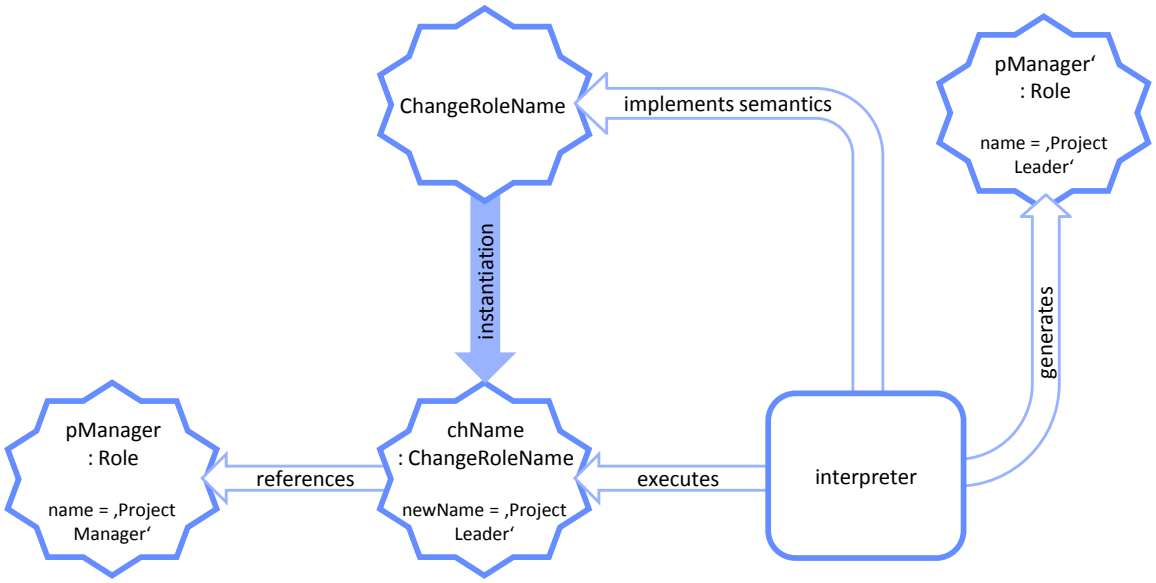

Figure 6.5: Exemplary modifiability scenario (legend: see FIGURE 6.2)

An interpreter is necessary to execute the change operations on the target elements.[8] The interpreter must know the semantics of the change operation types and be able to take into account additional parameters provided by the change operation's attributes.

Example Resolved. FIGURE 6.5 is showing an exemplary realization of the modifiability design approach. It's structure is deliberately orientated on FIGURE 6.4.
In the upper center, there is a change operation type denoted with *ChangeRoleName*. Below, an instance of this type is placed in the model: *chName*. It has an additional parameter *newName* storing some information to be used during interpretation of the change operation. *chName* references a model element within the domain model, namely *pManager*. This element has a *name* attribute with the value 'Project Manager'.
The described scenario can now be taken by an interpreter and transformed to a modified model. The semantics of the change have to be implemented by this interpreter. In the transformed model, placed at the right hand side of FIGURE 6.5, there is an element *pManager'*, where the *name* attribute's value has been replaced by the parameter of the change operation *chName*.

Variants. No variants could be identified.

Known Uses. The V-Model XT [BMI10] can be mapped to the design approach stated above. Its meta-model contains change operation types like *RolleUmbenen-*

[8] More precisely: to execute the *instances* of the former on the *instances* of the latter.

nen (RenameRole).
In addition, the provided tool suite of the V-Model XT supports the interpretation of the change operations made applicable by the meta-model. This interpreter uses an extension model and merges it with a reference model, thereby executing all change operations in the order of their appearance in the model. The result is separately saved in the file system for usage by other tools of the V-Model XT tool chain.

7. Development model lines

Contents

We can observe some efforts in the upcoming field of process line engineering. To take over the term *Process lines*[1] by learning from product line engineering was proposed by ROMBACH [Rom05]. He pointed out already existing examples of process lines,[2] and postulated the need for theoretical foundations.
A proposition for a concrete process line architecture was made by WASHIZAKI [Was06]. The concept is presented as a means to subsume several similar workflows under one template workflow that can be configured to the users needs. The paper concentrates on process descriptions and is providing a SPEM extension to express commonality and variability in process workflows. The approach is bound to SPEM's workflow model.
WASHIZAKI explains by means of an example how a set of process workflows can be integrated into a generic process pattern that can be used to create these workflows by serving variation points and selecting alternatives. In addition, this process pattern, denoted as a process line architecture, can be used to create many variants of the exemplary workflows.
The approach is an analytical approach taking into account a set of existing workflows, and creating a generic workflow that can be used to recreate all given variants, and more. The provided notation can even be used constructively to design generic

[1] For usage in the area of variable process models.
[2] The developers of these 'process lines' did themselves not necessarily use this term.

workflows that can be configured to the users needs, but does not provide a solution to the problems stated in CHAPTER 2. The reason is that it represents a specific solution for the configuration of workflows, whereas the stated problems reside in another level of relevance.
ARMBRUST *et al.* researched the scoping of process lines without providing concrete specifications for a process line architecture [AKM+09]. I.e., they propose an approach to analyze current and future products and projects in an organization to elicit its process needs, interpreting these needs as the scope of the process line [AKM+09, p. 185 f.]. The approach consists of five steps, namely product analysis, project analysis, process analysis, attribute prioritization, and scope determination using a mathematical model.
The given approach of a DML framework in this chapter is to be understood as a complementary addition, since scoping of development model lines is out of the focus of this thesis. The relevant aspect in this thesis is a structure of a development model that is intended to be configurable, extensible, and modifiable.
GOALS 1 and 4[3] represent the ambition of this thesis to support the theoretical foundations asked for by ROMBACH [Rom05].
This chapter subsumes the definitions made in the previous chapters and provides a realization of the DML concept presented in CHAPTER 5. This concept makes use of the variability mechanisms identified in CHAPTER 6.
The Chapter contains first a SECTION 7.1 identifying different roles that may be present during the development of a concrete DML. Afterwards, the DML concept is explained in detail by first describing the structure of a DML framework (SECTION 7.2) and then explaining how a DML architecture is created (SECTION 7.3). In SECTION 7.4, it is shown how such an architecture can be used to create a concrete DML. In order to enable configurability and modifiability of a DML, a DML environment is described in SECTION 7.6. SECTION 7.7 describes the case that a reference model is updated to a new version by the core asset engineer.
For more convenience while reading this thesis, certain DML specific keywords are marked with a short line printed above the keyword. As the DML concept described in this chapter differentiates between three different levels, for each keyword it is additionally indicated to which level the keyword belongs.

- 'contentElement' is an entity of the DML framework presented in SECTION 7.2. It is marked by a short line on the left: $\overline{\text{c}}$ontentElement.
- 'product' is an entity of the exemplary DML architecture presented in SECTION 7.3. It is marked by a short centered line: pro$\overline{\text{d}}$uct.
- 'projectManual' is an entity of the DML example presented in SECTION 7.4. It is marked by a line on the right: projectManua$\overline{\text{l}}$.

This kind of markings for entities defined at different levels is intended to retain clearness. For example, the three entities mentioned above can be related to each

[3] See CHAPTER 1.

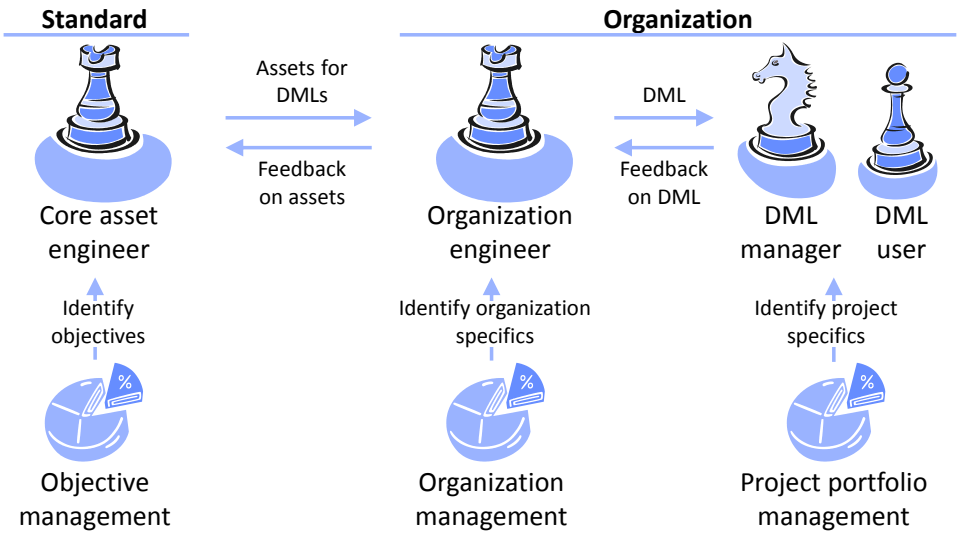

Figure 7.1: Roles of a development model line (adapted from [McG04, Figure 1])

other: contentElement is a generalization of product, while product is a generalization of projectManual. The intention of these markings is to provide a quick comprehension of the involved DML levels.
Marking entities has an additional advantage. When marked, a word can exactly be identified as indicating an actual entity. For example, in the DML framework, there is an entity relation. When used in a sentence within the scope of the DML framework, usage of the word 'relation' leaves it open if this word refers to a 'relation' between certain entities in a general sense, or to the entity 'relation'. Marking the word as relation definitely identifies it as an entity of the DML framework.

7.1. Developer roles in a DML

McGregor identified three roles that participate in the creation of a software product line [McG04, p. 66]. The first role is the group of *core asset developers*, which provides the resources needed to create products, such as the architecture and common system components. The group of *product developers* select appropriate assets from the core assets and create concrete products. The third role is the *management*, which both provides business objectives to be regarded during core development, as well as descriptions of the products that have to be created.
In the context of development model lines, similar roles can be identified, as shown in Figure 7.1. We observe that the role core asset developer and product developer exist there, too. To remain consistent with the nomenclature, they will be denoted with *core asset engineer* and *organization engineer*. Additional roles have been added, namely the *DML manager* and the *DML user*. Finally, the management has been subdivided into three separate roles.

The core asset engineer can use the DML framework presented in this thesis to create a domain-specific DML architecture. Using this architecture as a meta-model, the core asset engineer can create a reference DML. This DML can be used by an organization engineer to create customized variants of the DML. Such variants are translated to a final DML, which is taken by a manager for the creation of a project-specific development model. This specific model is taken by a DML user during execution of a project.

In comparison to the role model for software product lines presented by McGregor, the management part has been divided into three separate roles. This is due to an additional gap in the case of development model lines: core asset development and DML development usually are performed in different organizations with differing managements, whereas product lines usually are developed either within a single organization, or if multiple organizations are concerned, the management of the objectives and the target products exist under one organization.

So when having separate organizations involved, the management has to be divided. The division criterion is the function that is performed by the management. The three relevant functions are *Identify objectives*, *Identify organization specifics*, and *Identify project specifics*, in adaptation of the original functions [McG04, p. 66]. The roles to take up these functions are *objective management*, *organization management*, and *project portfolio management*.

In some cases, some of these management roles may be combined under one umbrella, resulting in a setting with a combined management like that described by McGregor. But in general, the roles have to be differentiated.

7.2. Structure of a DML framework

Figure 7.2 shows a UML diagram of a class structure for a DML framework as defined in Section 5.1.[4] The classes are divided into three groups: *framework core*, *knowledge pool*, and *feature model*.

The elements presented in this chapter enable a knowledge pool, as well as the properties of configurability, modifiability, and extensibility of development models.

Note that all elements except those in the feature model are abstract, i.e. they have to be specialized prior to usage. This specialization is made by a DML architecture.

The following sections describe the structural properties of a DML, beginning with a description of the DML framework. Afterwards, for this framework, an exemplary DML architecture is presented. This architecture is then used to describe an exemplary DML.

[4] In order to more easily distinguish classes from association classes in the UML diagrams, classes are drawn as a box filled with white, while association classes are colored blue.

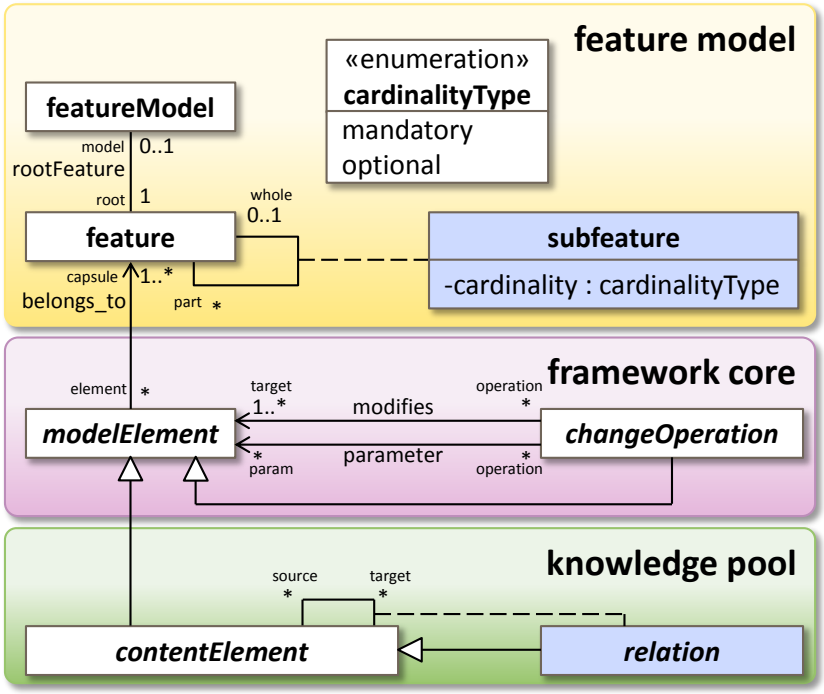

Figure 7.2: The elements of a DML framework

7.2.1. Framework core

This section covers the description of the framework core as illustrated in the middle part of FIGURE 7.2. The central element of a DML framework is the most generic class we can think of: the abstract class *modelElement*. When finally constructed, a development model only consists of instances of this class. The DML framework core does not imply any structural conditions for the meta-model in a DML architecture, with the exception that any instance of modelElement must be associated with at least one instance of feature (see SECTION 7.2.3).
The area covered by the *changeOperation* and its outgoing associations, is based exactly on the modifiability design approach presented in CHAPTER 6.3.2. Its structure is shown in FIGURE 6.4.
There is a 1:1 mapping possible between this approach and the DML framework described above. TABLE 7.1 is mapping the modifiability entities to their counterparts in the DML framework. Specializations of changeOperation to be defined within a DML architecture represent the *change operation types*, while their instances map to the *operation instances*. The *target element* is realized by instances of contentElement attached to any changeOperation via the modifies association.
Finally, the *interpreter* described in the modifiability design approach is realized by the DML environment. It is not shown in FIGURE 7.2, because it represents an external application logic. It must implement the semantics for all change operation types. Doing so, it is an integral part of the process to generate development models

Modifiability entity	DML entity
change operation type	specializations of c̄hangeOperation
operation instance	instances of c̄hangeOperation
target element	instances of c̄ontentElement referenced via m̄odifies
interpreter	DML environment (not shown in FIGURE 7.2, see SECTION 7.6)

Table 7.1: Mapping between modifiability design approach and the DML framework

and its implications map those of the interpreter described in the modifiability design approach. The DML environment is described in SECTION 7.6.

7.2.2. Knowledge pool

The class *c̄ontentElement* is a specialization of m̄odelElement. c̄ontentElement is the abstract class for anything that can be defined in a knowledge pool. Instances of c̄ontentElement can be related to each other by an abstract association *r̄elation*, which itself is a specialization of c̄ontentElement. Thus, it is part of the knowledge pool. Through c̄ontentElement and r̄elation, an arbitrary meta-model for the knowledge pool of a development model can be created as part of a DML architecture. This creation is made by specialization, as shown in SECTION 7.3.
Note that any instance of a subclass of c̄ontentElement belongs to the knowledge pool.
The usage of r̄elation is a quite inconspicuous realization of the extensibility design approach described in CHAPTER 6.2.2. The realization of r̄elation as an association class was explicitly chosen to give the creator of a DML architecture the opportunity to include extensibility aspects into the meta-model by specialization of this class. As we will see below in SECTION 7.4, an instantiated specialization of this class can be used to extend an existing model without applying direct changes to the original model during variant modeling.

7.2.3. Feature model

The configurability approach, shown at the top of FIGURE 7.2, is based on a feature modeling concept named *Forfamel* [AMS06, AM09]. This concept is realizing feature models as designed in FODA [KCH+90].
A *f̄eature* represents an end-user visible characteristic of a domain [KCH+90, p. 3]. Configuration of a development model is realized through selection of features. An example for a system composition using FODA has been illustrated in FIGURE 3.1.
A feature may consist of other features. This is modeled as an association *s̄ubfeature*, which is provided with a feature type. The possible types are defined by the enu-

meration *cardinalityType*:[5] mandatory and optional. A mandatory feature must be selected, if its parent feature is. An optional feature may be selected, but does not need to be.
Optional features provide configurability, as they defer the decision concerning the actual configuration of a development model to the DML manager.
A modelElement must be associated with at least one feature. This association *belongs_to* indicates that a modelElement is part of the resulting development model, if the parent feature is. A modelElement may be assigned to different features simultaneously. If any of these features are selected, the modelElement is selected, too.
Finally, the configurability area contains a class *featureModel*, which primarily is used to identify a root feature to begin development model configuration with. A feature that is related to the featureModel as *rootFeature* is mandatory to any development model that can be created using the DML.
Note that the configurability classes are final, which means they cannot be specialized in a DML architecture, but have to be instantiated directly in a DML.[6]
The described approach is an implementation of the configurability structure described in CHAPTER 6.1.2. The feature represents the whole, while a feature related to the whole via a subfeature association plays the unit. In our case at hand, units may in turn play the role as a whole.
The selection of features can be denoted with *tailoring*. As described in CHAPTER 6.1.2, a *Configurator* component is needed to enable this kind of tailoring. This component is realized by the DML environment. See SECTION 7.6 for details.

7.3. An exemplary DML architecture

A development model makes use of a meta-model describing the types of admissible model elements. Such a meta-model is denoted in this thesis as a DML architecture. As described in CHAPTER 5.1, a DML architecture is created by specializing the DML framework.
In the DML framework, all elements are declared abstract, except those of the feature model. I.e., prior to usage, they have to be specialized in a DML architecture.
In this section, we will take a look at a development model meta-model that has been created to serve as an example of a DML architecture. It will be used as a meta-model for the creation of a development model example in SECTION 7.4. FIGURE 7.3 shows domain-specific classes that specialize some abstract framework classes introduced in SECTION 7.2.[7]

[5] Note that the name 'cardinalityType' is adopted from [AM09] for consistency reasons.
[6] In UML, this property is specified by setting a *RedefinableElement*'s attribute *isLeaf* to 'true'.
[7] The shown model makes use of subsetting association ends, but without renaming the ends. See discussion in CHAPTER 11.1.1.

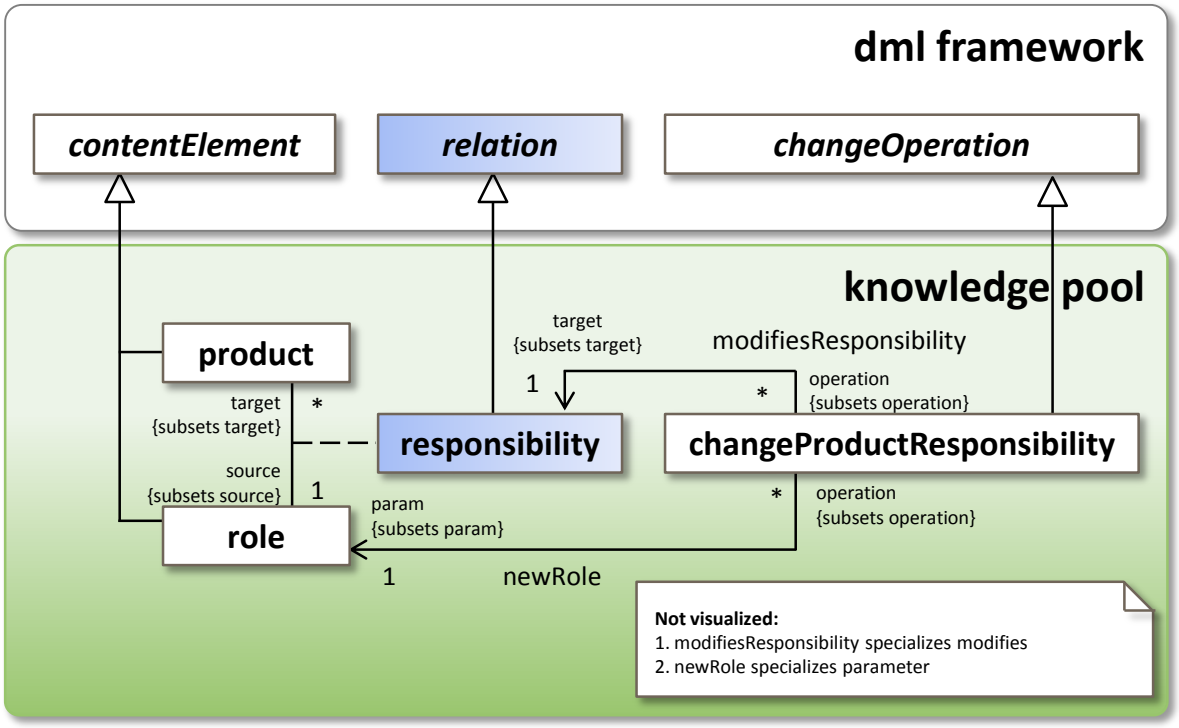

Figure 7.3: Specific elements of an exemplary DML architecture

On the left hand side, we can see two typical specializations of contentElement: *role* and *product*. In addition, there is a specialization of relation called *responsibility*, which connects each role with an arbitrary amount of products.[8]

Clearly, the role element addresses the *role model* reference area in a development model, whereas the product element is concerned with the *product model*. See CHAPTER 2.1.2 for details on the reference areas.

In addition, there is a specialization of *changeOperation* that was given the name *changeProductResponsibility*. This specific change operation has an outgoing association *modifiesResponsibility* to a responsibility-relation and an outgoing association *newRole* to a role.

In contrast to the responsibility specialization, the exact semantics of this class is important to us. Each selected[9] instance of a change operation has to be operationalized by the DML environment. How the semantics of a change operation can be specified is shown in CHAPTER 10.3.

In case of an instance of changeProductResponsibility, the DML environment will

[8] There is no specification of a detailed semantics for this relation here, because this is not needed in this example. We may simply assume any semantics that appears to be reasonable. Note that in a real development model, the semantics of each relation between contentElements must clearly be stated!

[9] I.e., it belongs_to a feature that has been selected during configuration of a DML.

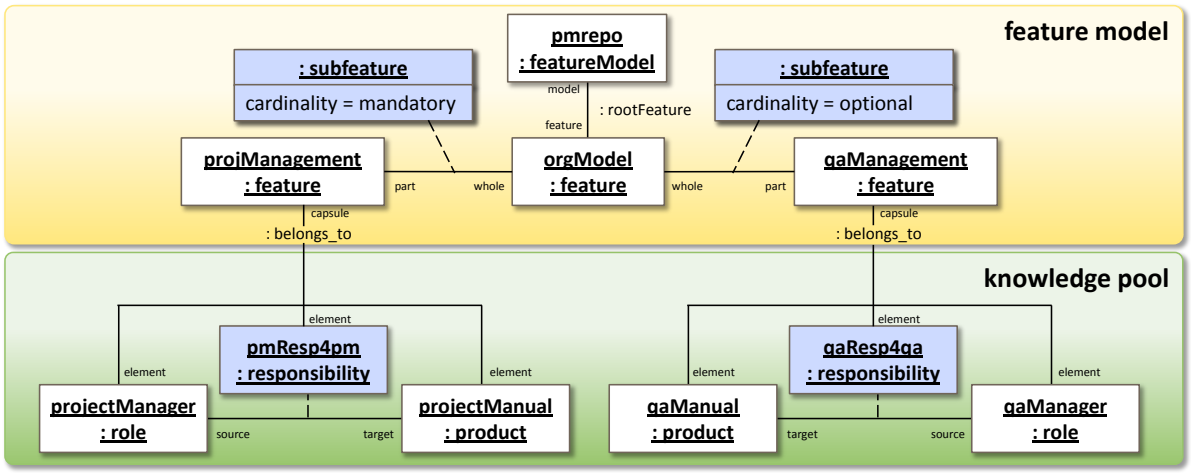

Figure 7.4: Exemplary DML element instances

modify the responsibility identified by the association *modifiesResponsibility*. After execution of the operation, the responsibility is related to another role, namely the role referenced by the *newRole*-association of the changeProductResponsibility instance. See SECTION 7.4 for an instance example of changeProductResponsibility and its implications.

7.4. An exemplary DML

In this section, we will take a look at an exemplary DML created using the DML architecture described in SECTION 7.3. The example is illustrated by FIGURE 7.4.
As stated in CHAPTER 5.1, a DML is created via instantiation of a DML architecture. On top of FIGURE 7.4, we can see instances of the final classes in the feature model that were defined in the DML framework. In addition, we observe instances of classes defined in the DML architecture, belonging to the knowledge pool.
In the feature model area, there is a featureModel *pmrepo* and three features, namely *orgModel*, *projManagement*, and *qaManagement*. orgModel is related to pmrepo and thus indicated as the root feature in the model. This results in orgModel being a feature that is selected in any configuration. projManagement and qaManagement are subfeatures of orgModel. projManagement is a mandatory feature, while qaManagement is modeled as optional via the subfeature association. The described feature model allows for two possible configurations of the DML: in the first configuration, only orgModel and projManagement are selected, and in the second, all three are selected. This feature model spans the configurability of the DML.
As we can see in the DML framework, all modelElements, and thus all elements within the knowledge pool, must be related to an instance of a feature via an

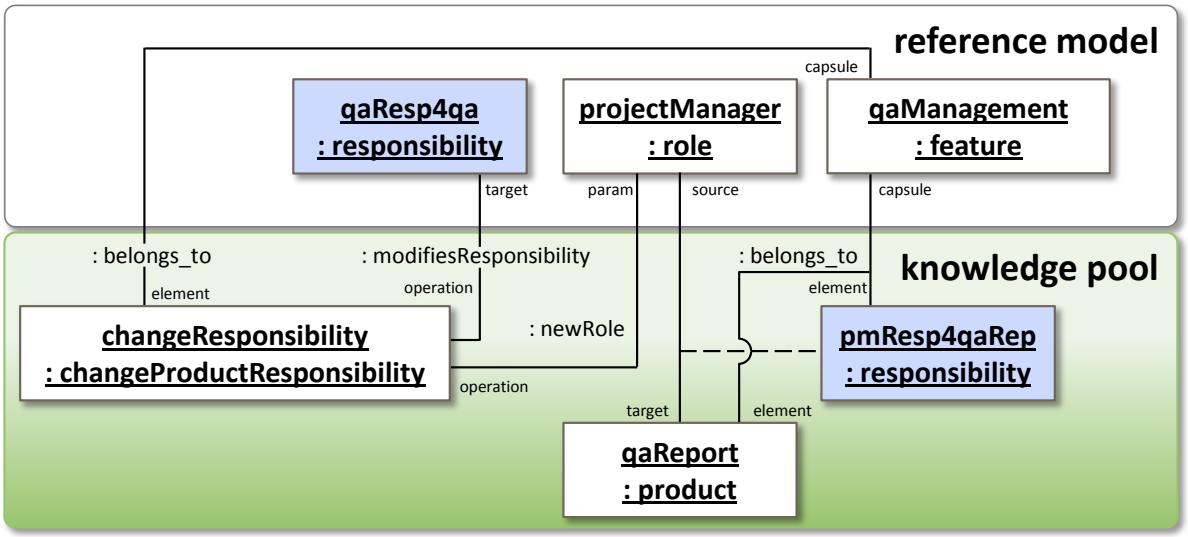

Figure 7.5: Exemplary organization-specific DML extension

instance of the belongs_to association. We can see that this is the case in FIGURE 7.4. For each of the two dependent features, both a product and a role are contained. For projManagement, there is a role *projectManager* and a product *projectManual*. For qaManagement, there is a role *qaManager* and a product *qaManual*. These pairs each are related by a responsibility-association.
Note that responsibility is a specialization of relation, which is a modelElement. This implies that instances of responsibility, like all other modelElements, must have a belongs_to association to a feature.

7.5. An exemplary DML extension

FIGURE 7.4 illustrates a reference DML, being the root model for a variety of development models that may be deduced from it using configuration, extension, and modification. The configurability of the reference model is defined by the feature model. FIGURE 7.5 is showing an example making use of both extensibility and modifiability. It shows an extension model that may be created by an organization to create a customized variant of the reference model. This extension model is making use of the very same meta-model as the reference model, which is the DML architecture.
The example in FIGURE 7.5 shows that a product instance *qaReport* has been added. The projectManager located in the reference model is responsible for qaReport via a responsibility instance. This is a pure extension to the reference model. The extension is made possible by the fact that responsibility is a relationship object, instead of an attribute of product or role. This allows for the creation of relations between elements, even if these elements are located within the reference model,

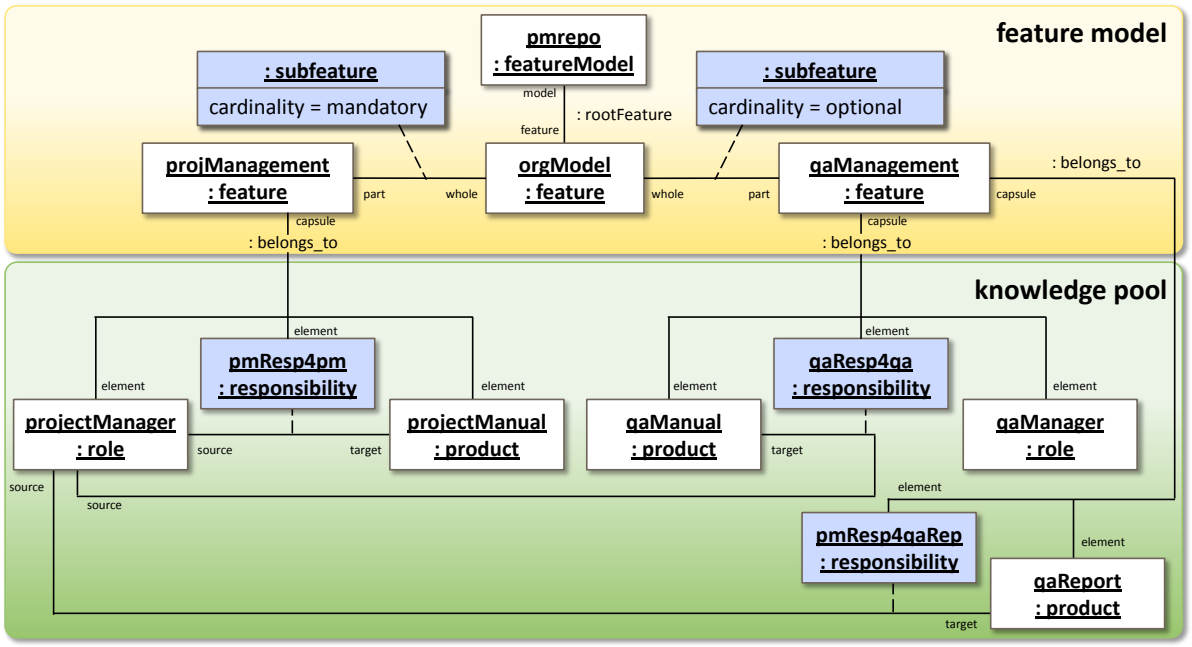

Figure 7.6: An interpreted development model variant

without performing changes to the reference model itself.
In addition to the mentioned extension, the customized model contains an instance of changeProductResponsibility named *changeResponsibility*. This change operation belongs to the feature qaManagement, which means that the operation applies to all configurations of the model, where this feature is selected.
The change operation is pointing via the association modifiesResponsibility to the responsibility *qaResp4qa*. qaResp4qa is a relation between the product qaManual and the role qaManager in the reference model. This responsibility is the target of the change operation, i.e. this is the element where the modification has to take place. A parameter newRole is associated to changeResponsibility, too. This association relates it to the role projectManager. Thus, the reference to the projectManager is the needed additional information for the change operation to know which role has to be inserted into the responsibility qaResp4qa.
The modification itself is executed by the DML environment after configuration of the DML. The semantics of the change operation implies that after execution of the operation, the responsibility instance qaResp4qa is no longer related to the role qaManager, but to the role projectManager.
The result of the described models after interpretation by a DML environment is shown in FIGURE 7.6. We can see that the responsibility qaResp4qa is now related to projectManager. This is due to the change operation changeResponsibility. Before interpretation of this operation, qaResp4qa was related to qaManager. In addition, there are the newly added instances *qaReport* and pmResp4qaRep.
Note that the figure is showing the development model after configuration, exten-

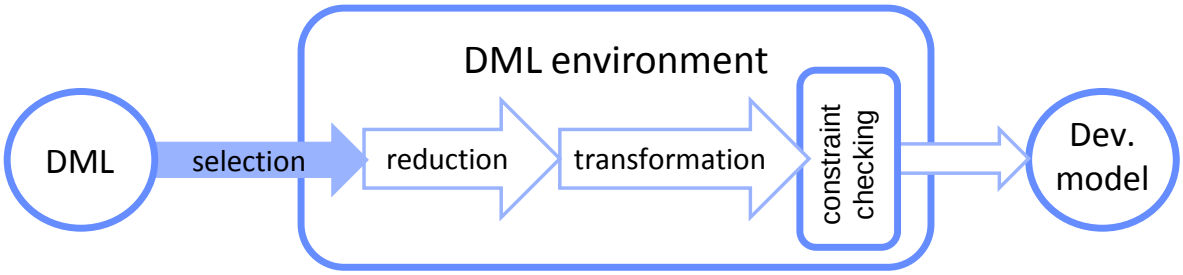

Figure 7.7: Process steps of a DML environment

sion and modification. I.e., in the example shown, the feature qaManagement has been chosen by the user triggering the DML environment. Otherwise, the change operation would not have been interpreted, as it belongs_to qaManagement.

7.6. DML environment

In the creation process of a development model, a DML environment is performing automated transformation steps to create a development model from a DML. As indicated in FIGURE 7.7, a DML environment is used for two model transformation steps: first, reduction according to given configuration data, then transformation by execution of change operations.
The combined process of configuration and reduction is called *tailoring*.
FIGURE 7.7 is actually a detailed view of the translation of a DML to a development model, as it was shown in the lower right part of FIGURE 5.1.
After the creation of a reduced and transformed development model, automated result analysis by checking constraints is performed. A transformation is only successful if the result analysis yields a positive result.
The reduction and transformation of DML's are modeled using the DML entities features and changeOperations, respectively. For result analysis, constraints will be introduced in CHAPTER 9.
The following paragraphs contain descriptions explaining the purposes and execution strategies of reduction, transformation, and constraint checking in the context of DML's. These descriptions can be seen as informal requirements a DML environment must implement. An exemplary implementation example is provided in CHAPTER 10.

Reduction. To realize configurability, a DML environment must provide a selection mechanism for a DML manager to enable configuration of a DML by selecting features. For this purpose, a semantics is provided for the association *belongs_to* we know from the DML framework.
A feature that is bound to its parent feature via a mandatory subfeature relation is always selected, if its parent is. A feature bound via an optional subfeature relation

may be selected if a user selects it.
Every $\bar{m}$*odelElement* that does not belong_to[10] a selected $\bar{f}$eature is removed from the DML and will not be part of the resulting development model.

Transformation. After the reduction, if there are any $\bar{c}$*hangeOperation*s in the resulting development model, the DML environment executes each operation by operationalizing its semantics defined along with the DML architecture (see CHAPTER 10.3).
As a basis, and for simplicity, a total order over the change operations will be assumed. This order has to be modeled in the DML. It is not an aim of this thesis to provide a more thoroughly designed approach for operation sequencing. For simplicity, it is assumed that change operations are executed in the order they are modeled in the persistent file containing the model, exactly like difference operation ordering is specified in XMI [OMG07a, p. 38].

Constraint checking. The result of the previous two steps is then checked against constraints. These constraints are provided in two levels. The first level is implemented in the DML architecture, where a set of domain-specific $\bar{c}$onstraint specializations is defined for a particular DML. The second level is implemented in a DML, where the $\bar{c}$onstraint specializations are used as meta-elements for concrete constraint instances. A model element can be related to a constraint instance in order to determine whether the contextual environment of this element adheres to given constraints.
Along with each $\bar{c}$onstraint specialization, a concrete semantics must be provided so that the DML environment is able to decide whether a constraint instance is fulfilled.
If the model resulting from the transformation step is not contrary to any constraint, the result is a development model ready to be used for a project-specific context by the DML user.
For the sake of simplicity, $\bar{c}$onstraints have not yet been included in the DML framework. An introduction to $\bar{c}$onstraints is provided in CHAPTER 8 and the integration of constraints with the DML framework is described in CHAPTER 9.

7.7. Upgrading the underlying DML

The DML architecture presented in SECTION 7.3 gives the organization engineer of an organization-specific adaptation the possibility to exchange the standard DML with a newer version. The consequences are explained by means of an example in this section. We assume that the creators of the DML model from the example in SECTION 7.4 publish a new version of the DML.

[10]The association name has been flexed from $\bar{b}$elongs_to to belong_to for better readability.

Let's assume that compared to its previous version, there are two differences in the new version. The first is the renaming of the role projectManager to projectLeader.[11] The second change is the addition of a new mandatory feature with the name configurationManagement. This feature contains the newly created product instance *configurationHandbook* and assigns the role projectLeader as being responsible.
When the organization engineer of a customized model variant intents to upgrade his model to the new version of the standard DML, he must exchange the persistent file containing the original model with the new one. I.e., the extended customized model no longer references to the old version, but to the newly adapted standard DML.
The organization engineer now looks at the new contents for any conflicts that may arise when including the new model. There are no conflicts,[12] so all he has to do is to exchange the old model with the new model. Every project initiated from now on will be based on a new customized variant of the adapted standard DML, which inherently contains all features of the newest model.

[11] Note that while performing this change directly in the original model, the unique identifier of the altered class remains the same.
[12] We will take a look at possible conflicts in SECTION 11.4.

8. Concept for constraint restriction

Contents

In this chapter, we will take a look at a typed, model based mechanism for constraint restriction that relies on domain-specific rules. This mechanism is based on constraints that can explicitly be modeled within a DML. Constraints are a formal mechanism that can be used to test if a model satisfies exact requirements. Each constraint has a representative class in the DML architecture, hence model based. Constraints can be combined and evaluated using terms with a boolean logic.
With a DML architecture defining concrete constraints, it is possible to create a DML containing instances of these. Constraints may be linked to other elements within the DML.
When using this mechanism, we can define for any element in the DML that it must conform to a particular constraint, or a combination of constraints. As these constraints are defined formally, it is possible to automatically check the element's model environment for consistency to the constraints.

8.1. Motivational aspects

As we can see when taking a look at OCL, constraints are motivated by the need for invariants, preconditions and postconditions [MC99, p. 854]. Preconditions and postconditions are not covered in this thesis as they are concerned with operational transitions. Constraint restriction is defined to target a variability subject in its varied state.
In order to be able to apply constraints, the model context to be regarded during interpretation of a constraint must be supplied. This context can be either related to a concrete model element instance that is bound to a particular constraint, or to a type of model elements, indicating that the constraint is stated over all instances of this type. This motivates the distinction between two classes of constraints: $\bar{c}$*onstraint* and $\bar{t}$*ypeConstraint*. TABLE 8.1 gives an overview of these classes.

	c̄onstraint	**t̄ypeConstraint**
Target(s):	model element(s)	model element(s) and model element type
Modes:	(no modes)	any, all
Informal semantics:	The target(s) must fulfill specific criteria.	The target(s) and instances of the targeted type must fulfill specific criteria.

Note: The concrete criteria that must be fulfilled depend on the specialization created for constraint instantiation.

Table 8.1: Constraint types

t̄ypeConstraints are similar to what can be expressed with OCL. The constraint places the context upon a concrete type and states the invariant that has to be fulfilled. In OCL, such a constraint may be:

```
context Person inv: self.Age >= 0
```

This exemplary constraint states that any instance of *Person* must have an age greater than or equal to 0 at any time.
On the other hand, c̄onstraints are out of the focus of OCL, in regard to the constrained entity. c̄onstraints can be related to a concrete model element instance in a model, which is not within the expressibility of OCL. I.e., a c̄onstraint can be used to target an element and define the conditions this particular element must fulfill. This is like defining an explicit constraint over *Peter*, instead of *Person*.
t̄ypeConstraint is a specialization of c̄onstraint. I.e., it has the same properties but is additionally provided with a reference to a type to be constrained.
The targets of a c̄onstraint are actual model elements, i.e. they are bound variables. This is unlike the usage in OCL, where parameters are always free variables. That way, a constraint instance may be formulated concerning a particular model environment.
For t̄ypeConstraints, it is important to specify the environment in which the constraint is fulfilled. The constraint may be satisfied if *any* instance of the targeted type satisfies the constraint, or it may only be satisfied if *all* instances of the type fulfill the constraint.
Note that modeling constraints as classes implies that constraints are placed within a model by instantiating the appropriate class and setting target values. I.e., they are no static part of the meta-model, but part of the model itself with the semantics being stated within the meta-model level.
The classes c̄onstraint and t̄ypeConstraint are unspecific in their concrete semantics. The core asset engineer must create specializations of these classes during the creation of a DML architecture. Along with each specialization, concrete semantics must then be provided. How this can be realized is shown in CHAPTER 10.4.6.
In addition to these two constraint classes, it appears advantageous to be able to combine different constraints with each other, in the sense that we can create terms of constraints for an increased flexibility to express constraints.

These two concepts, namely the differentiation between the two constraint types constraint and typeConstraint, and the possibility to create constraint terms are the basis for the constraint mechanism explained in this chapter.

8.2. Constraint restriction example

Let's remember the DML architecture defining the contentElements product, role, and the relation responsibility between role and product, as was shown in FIGURE 7.3.
The following instances that were shown in shown in FIGURE 7.4 will be subject to the constraint example.

- projectManual, instance of product,
- projectManager, instance of role,
- qaManual, instance of product,
- qaManager, instance of role,

As creator of a reference DML, i.e. as core asset engineer, we might want to define restrictions for all variants created from the reference model. In our example, we assume a restriction for products. We want to express that...

... in all modified variants of the DML, the projectManager is responsible for the product projectManual.

... if the qaManager is responsible for any product, then he must at least be responsible for the qaManual.

We want to define this requirement at the instance level, since otherwise we would have to integrate this particular circumstance into the DML architecture, where, in the first line, we do not know what instances will be defined in the DML.
Now, there are two analytical approaches suitable to make sure that a model is consistent with the domain-specific restrictions listed above. If this restriction was only defined using natural language, like it is up to now, the first approach would be to perform a manual check.
The second approach, however, requires a formalization of the constraint, as well as an engine able to map the formal specification against the implementation in the concrete model. This approach will be dealt with informally in this section. A formal basis will be provided in CHAPTER 10.
A meta-model for the example described below is presented in FIGURE 8.1. This example is a reduced version of the concept that will be introduced in CHAPTER 9. It does only show how constraints may be specialized in a DML architecture.

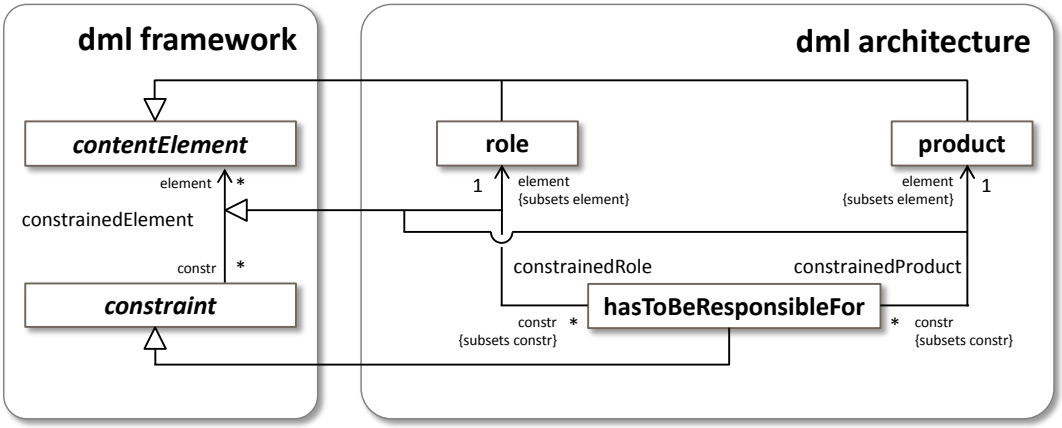

Figure 8.1: Example for a constraint meta-model

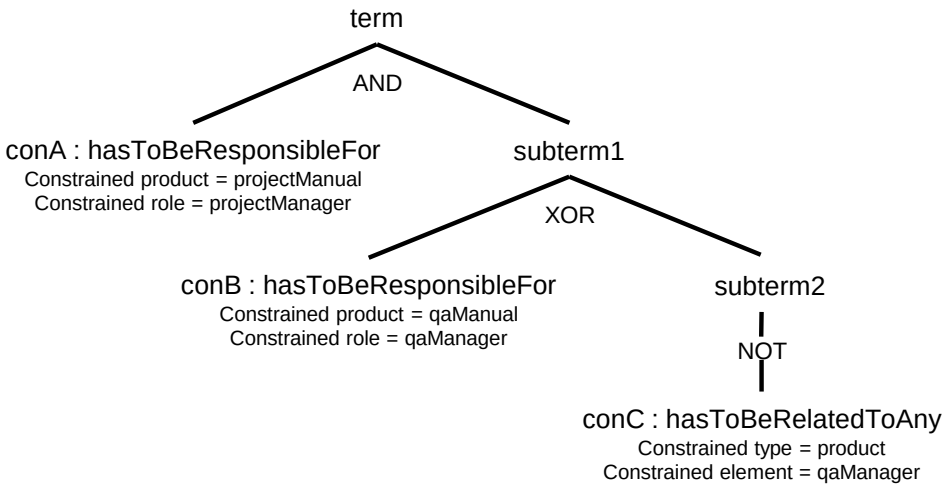

Figure 8.2: Exemplary constraint term

hasToBeResponsibleFor specializes constraint. The contentElements that are target of the constraint are bound to hasToBeResponsibleFor by its outgoing associations: *constrainedRole* indicating a role, and *constrainedProduct* indicating a product.

Note that the associations both specialize constrainedElement. All involved association ends are subsetting the association ends element and constr.

The informal semantics of *hasToBeResponsibleFor* is that an instance of this class is deemed an unsatisfied constraint, if there is no responsibility relation between the targeted instances of role and product.

FIGURE 8.2 shows an exemplary term consisting of three constraints. The constraints are named *conA*, *conB*, and *conC*. The two constraints *conA* and *conB* are instances of hasToBeResponsibleFor, a specialization of constraint. *conC* is an

instance of hasToBeRelatedToAny, which is a specialization of typeConstraint.[1] These three constraints are combined to a term consisting of constraints and subterms.
conA is a constraint over the product projectManual. It ensures that there exists a responsibility relation between projectManual and projectManager. This constraint is associated to *subterm1* with a boolean AND operator.
subterm1 consists of two operands, namely *conB* and *subterm2*. These operands are combined with an XOR operator. *conB* is a constraint that is satisfied if qaManager is responsible for qaManual. If this is not the case in a model in its varied state, then *subterm2* must be satisfied.
subterm2 negates the associated constraint *conC*. *conC* is satisfied, if qaManager is related to any product instance. I.e., *subterm2* is only satisfied, if qaManager is not related to any product in any way. In effect, a model may only be valid if qaManager is either not related to any product, or qaManager is responsible at least for the qaManual.

8.3. What has not yet been investigated

There reside several aspects that need to be clarified. First, we only had a glimpse on the meta-model that is involved for constraint restriction, the complete constraint mechanism is yet to be defined. This point is taken into CHAPTER 9, where the whole constraint mechanism is integrated into the DML framework.
Second, the description is informal. A formal clarification of the evaluation mechanism and the rule semantics is provided in CHAPTER 10.
Third and last, it has not yet been fully motivated why to refrain from an existing constraint language like OCL. An explanation follows here.
To begin with, we have to recall the context in which DML's are to be used. It is all about variability of a development model standard. When an organization decides to create a customized variant, it is confronted with three actualities it does not have much influence on: a DML framework as it is proposed in this thesis, a finished DML architecture, and a concrete DML. The two last items are created by the core asset engineer.
The organization engineer can use the DML as a reference model and create a variant by usage of configurability, extensibility, and modifiability. While doing so, the organization engineer has to respect the restrictions modeled in the reference DML. I.e., if the core asset engineer creates rules as indicated in the example above, he can guarantee that in every valid variant of this DML, the projectManager is responsible for the projectManual and the qaManager is either responsible for the qaManual, or not responsible for any product at all.
Now the point is, the core asset engineer can define what requirements a model has

[1] hasToBeRelatedToAny will be included into the DML architecture in CHAPTER 9.

to fulfill in order to be a valid variant of the original model. This definition is made at the instance level of a DML, not the DML architecture. The DML architecture must only provide a generic structure to do this. This means, in the DML architecture, there must be classes defined to instantiate constraints, as well as their exact semantics. But the combination of constraints in respect to concrete instances within the DML is only made in the reference DML.
For the organization's organization engineer, this means he can uniformly use modeling elements of the DML architecture without knowledge of OCL constraints. In addition, he can add own restrictions, as well, having the possibility to model constraints over any variant of the customized DML.

9. Constrained development model lines

Contents

This chapter presents the complete constraint restriction mechanism that was introduced partially in CHAPTER 8.
The classes c̄onstraint and *t̄ypeConstraint* will be described in detail, as well as the possibility to combine these to build terms.

9.1. DML framework with constraint restriction

FIGURE 9.1 shows an extended version of the DML framework that was introduced with FIGURE 7.2. As part of the knowledge pool meta-model, meta-model elements have been added for constraint restriction. Two of these elements have already been mentioned in CHAPTER 8.2: *c̄onstraint* and *t̄ypeConstraint*.
The constraint restriction meta-model in its full extent additionally contains the following DML framework entities: *c̄onstraintOperand*, *ō perand*, *t̄erm*, and *r̄ootTerm*.
ō perand is the most generic class in the constraint restriction mechanism. A DML environment must be able to check whether an ō perand is fulfilled or not. How the DML environment can deduce such a decision is dependent on the actual specialization of ō perand that is instantiated in the model. Possible specializations can belong to one of two classes: t̄erm or c̄onstraintOperand.
The two aspects of constraint restriction identified in CHAPTER 8.1, namely the definition of constraints and the definition of terms consisting of constraints are covered in the following sections.

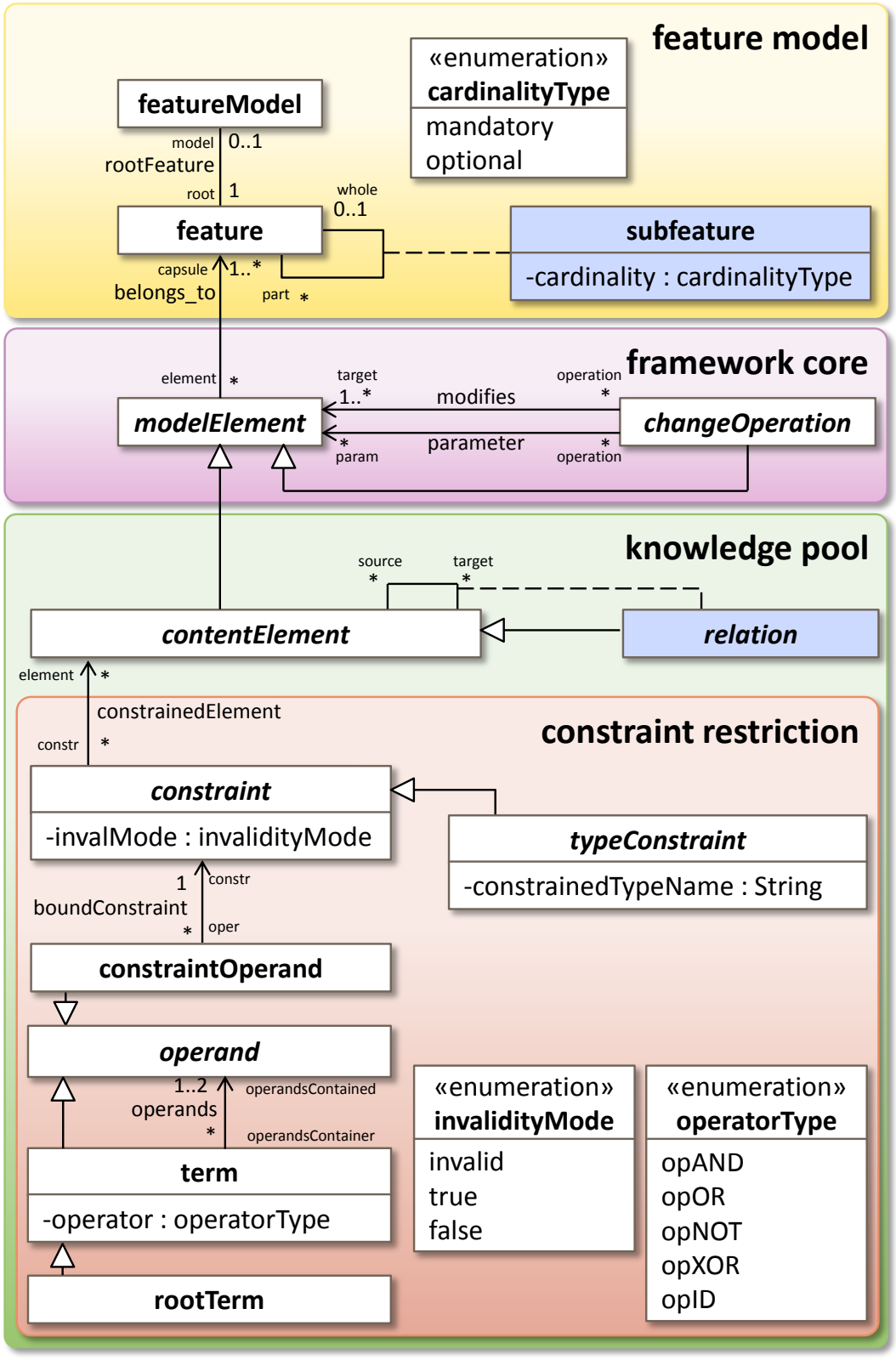

Figure 9.1: Constraint enabled DML framework

9.1.1. Constraints

A constraint is used to express an invariant over a model. The two types of constraints introduced in CHAPTER 8.1 are constraint and typeConstraint.
Any instance of constraint represents a restriction formulated over contentElements identified by the association *constrainedElement*. During evaluation of the constraint, the associated contentElements are tested against domain-specific requirements.
In addition to its references to constrainedElements, a typeConstraint has a string attribute *constrainedTypeName*. This attribute is used to specify the name of a type that is to be constrained.
When specifying the concrete semantics for typeConstraint in one of its specializations, it must be clarified whether the constraint is fulfilled when *any* instance of the associated type fulfills certain requirements, or *all* instances of the type must fulfill these requirements. This differentiation has to be made explicit as a specification of constraint semantics during realization of a DML architecture.
constraint has an attribute *invalMode* that has not yet been mentioned. Its purpose is to support the constraint restriction mechanism to be able to operate on reducible models. I.e., a model element targeted by an constraint may be removed from the model during model reduction. In such a case, the semantics of a constraint has been undefined until now. invalMode can be set to one of three values, with the following semantics. These semantics apply if the constraint instance has no constrainedElement set:

- **invalid**: The whole term checking for this constraint must evaluate to false.
- **true**: The constraint evaluates to true, always.
- **false**: The constraint evaluates to false, always.

Note that there is no value 'valid', always resolving a term to be true. This is because it would break the locality of a constraint. Of course, invalid breaks with this locality principle, too, but is regarded as a proper mechanism to deal with invalid term structures. This relation can be compared with the relationship between a process returning an error when a subprocess cannot successfully be performed in opposition to a process returning success when a single subprocess was performed successfully. In most cases, the latter behavior would be regarded as faulty.
Until now, it has been left open how a DML environment can determine whether a particular constraint is satisfied. For each specialization of constraint in the DML architecture, specific semantics must additionally be provided to enable constraint restriction. How such semantics can be defined is described in CHAPTER 10.
In the following section, we assume that the DML environment is able to decide for each specialization of constraint in the architecture whether it is fulfilled by a concrete DML.

9.1.2. Terms

In order to provide more expressivity for the constraint mechanism, a boolean logic term principle was added to the concept. This mechanism is represented by the *term* class.
The DML environment must be able to test whether a term is satisfied. A term is satisfied if all operands related to the term via the association *operands* are satisfied according to the *operatorType*.
Since term is a specialization of operand, the DML environment is able to interpret this kind of operand.
Another specialization of operand is *constraintOperand*. This class creates an attachment between terms and constraints via the association boundConstraint. operand itself does not have the association boundConstraint to avoid its specialization term to be able to directly be associated to a constraint instance. This relationship has to be realized as an operand instance.
A DML environment evaluates a constraintOperand by evaluating the associated constraint.
operatorType defines the boolean operators AND, OR, XOR, NOT, and ID, represented by the literals *opAND*, *opOR*, *opXOR*, *opNOT*, and *opID* respectively.
The semantics of these operators are as follows:

- **opAND** (binary): The term is evaluated with the result 'true' if and only if all operand instances associated via operands are evaluated to the result 'true'.
- **opOR** (binary): The term is evaluated with the result 'true' if at least one operand instance associated via operands is evaluated to the result 'true'.
- **opXOR** (binary): The term is evaluated with the result 'true' if and only if both of two conditions are given. First, the term must have exactly two operand instances associated via operands. Second, the result of both operands must be evaluated with a contrary result. I.e., if the first operand is evaluated to 'true', then the second must be evaluated to 'false', and vice versa.
- **opNOT** (unary): The term is evaluated with the result 'true' if and only if the operand instance associated via operands is evaluated to the result 'false'.
- **opID** (unary): The term is evaluated with the result 'true' if and only if the operand instance associated via operands is evaluated to the result 'true'.

For binary operators, the amount of operands related to a term must be two, for unary operators, the amount must be one.
A specialization of term is rootTerm. When checking a model for compliance with all instantiated restrictions, the DML environment has to evaluate all rootTerms and their subterms, and finally all operands associated via the association operands. There is no objection against including rootTerms as operands in another term.

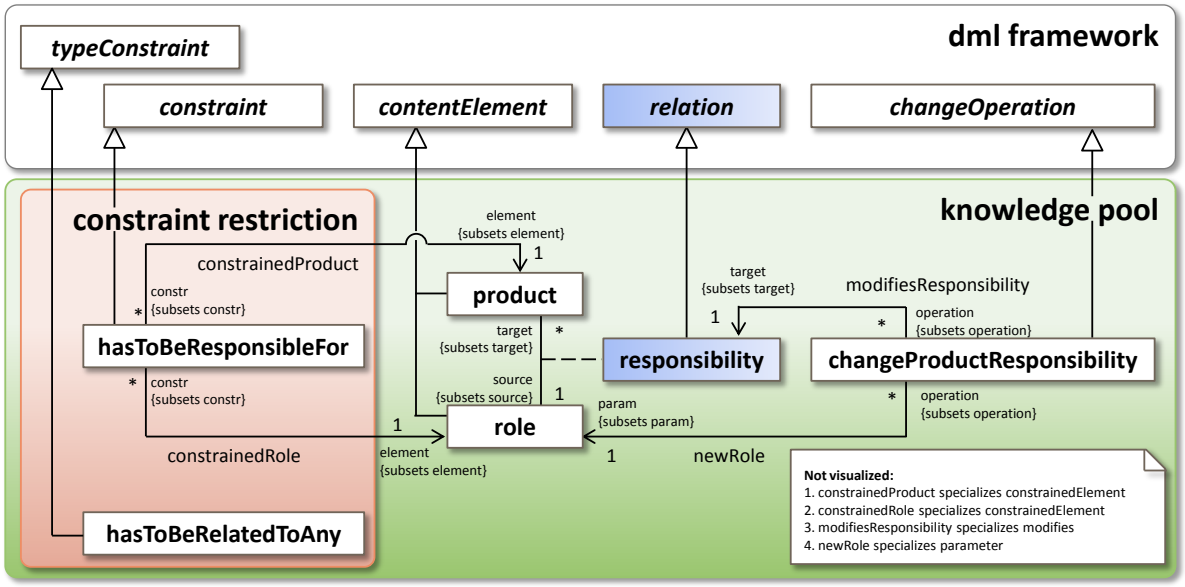

Figure 9.2: Example of a constraint enabled DML architecture

9.2. Exemplary DML architecture with constraint restriction

The previous section contained an extension of the DML framework presented in CHAPTER 7. Likewise, FIGURE 9.2 describes an extension of the exemplary DML architecture that was introduced by FIGURE 7.3. The extension contains specializations of constraint and is shown in a red box at the left hand side of the figure.

hasToBeResponsibleFor is a specialization of constraint. This class has already been introduced in CHAPTER 8.2.

It has an outgoing association *constrainedRole* which specializes the association constrainedElement. It is pointing to exactly one role. In addition, it has an association *constrainedProduct* that is specializing constrainedElement, as well, but pointing to a product.

An instance of hasToBeResponsibleFor represents a constraint checking whether a concrete product instance is responsible for a concrete role instance. We will see a model example in the next section.

Remember that the exact semantics of hasToBeResponsibleFor is discussed in CHAPTER 10.

In CHAPTER 8.2, another specialization of constraint has been neglected. In particular, it is a specialization of typeConstraint: *hasToBeRelatedToAny*. This specialization does not add any specialized associations to typeConstraint. Instead, the specialization is needed to attach semantics to this constraint.

The informal semantics of hasToBeRelatedToAny is that an instance of this class is deemed a satisfied constraint if any element associated to this instance via

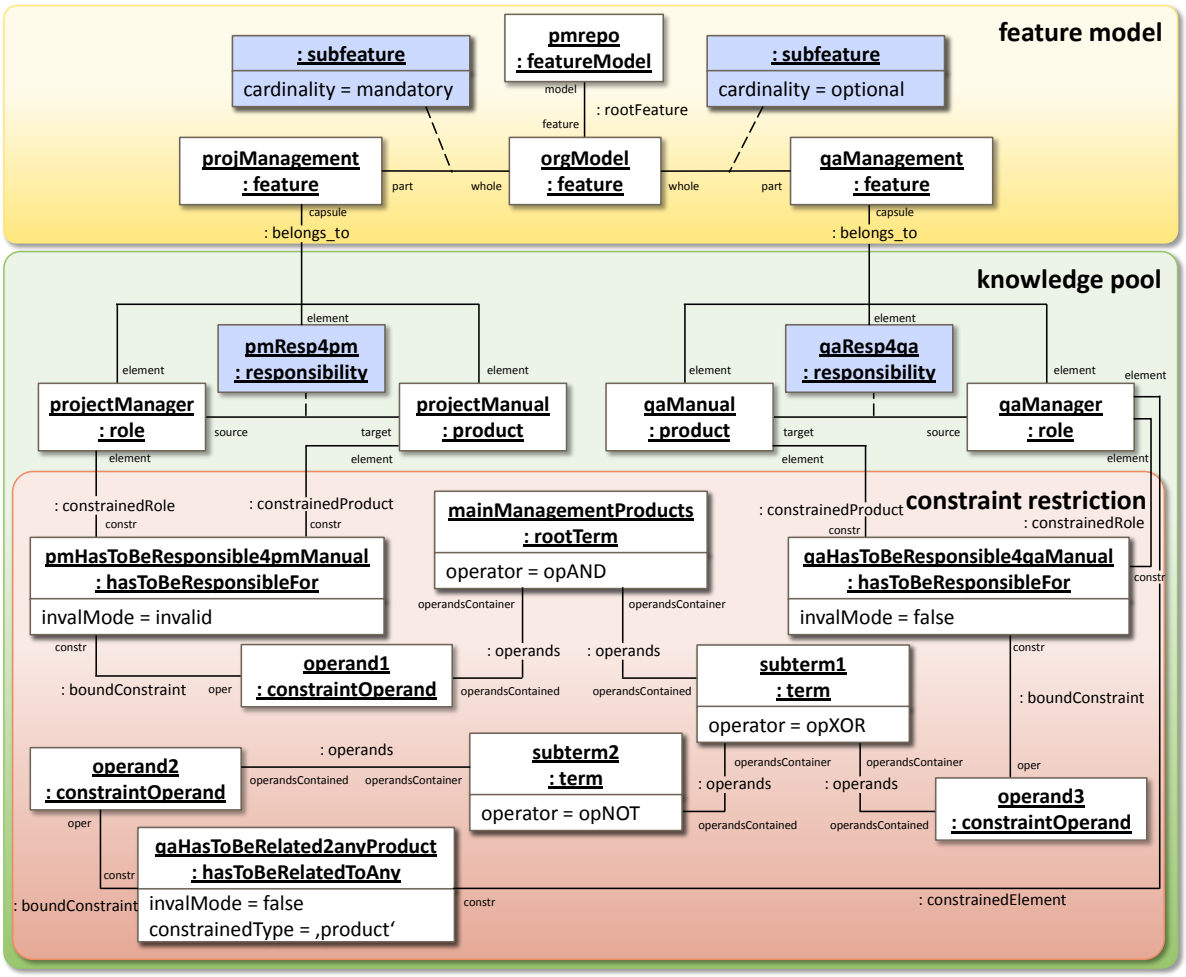

Figure 9.3: Example of a constraint enabled DML

its relation constrainedElement is associated to an instance of relation, which itself is associated to any instance of the class identified by the attribute value of constrainedTypeName. The attribute constrainedTypeName is inherited from typeConstraint.

9.3. Exemplary DML with constraint restriction

First of all, FIGURE 9.3 includes all DML elements that were presented in the exemplary DML introduced in CHAPTER 7. This was made to provide a complete example in this chapter, now including constraint restriction elements, too.
The constraint restriction example shown in FIGURE 9.3 has already been explained in SECTION 8.2, but not in all details.
The model placed in the constraint restriction box in FIGURE 9.3 is a DML representation of the term shown in FIGURE 8.2. mainManagementProducts is the term

FIGURE 8.2 illustrated. It is instantiating rootTerm. Any rootTerm is to be checked by the DML environment whether its operands fulfill the given rules. If any rootTerm is not fulfilled, the model is to be interpreted as an invalid model variant. A term that is not a rootTerm may not be fulfilled, but this does not necessarily mean the model variant is invalid.

In the example, the effect of the rootTerm is giving the DML environment the command to always check if the model created by configuration and modification is valid in respect to mainManagementProducts.

mainManagementProducts is deemed satisfied if both operands associated to the term are satisfied. That both operands have to be fulfilled is indicated by the value of operator: opAND.

One operand is operand1, an instance of constraintOperand. operand1 is associated to pmHasToBeResponsible4pmManual via the boundConstraint association. pmHasToBeResponsible4pmManual instantiates hasToBeResponsibleFor, which is a constraint specialization. I.e., a DML environment is able to decide whether pmHasToBeResponsible4pmManual is satisfied.

In our example, pmHasToBeResponsible4pmManual is satisfied if the role associated via constrainedRole is attached to the product that is associated via the relation constrainedProduct. In the model at hand, this constraint is fulfilled by the responsibility instance pmResp4pm: the role projectManager is responsible for the product projectManual.

We ignore hasToBeResponsibleFor's attribute invalMode for now. It will be covered below.

The other operand of mainManagementProducts is subterm1, which is another term. It consists of two operands subterm2 and operand3. Their satisfaction state is mutually exclusive (opXOR).

operand3 is provided with qaHasToBeResponsible4qaManual. This constraint is similar to pmHasToBeResponsible4pmManual and tests whether the qaManager is responsible for qaManual.

subterm1's other operand subterm2 uses the unary operator opNOT. This indicates that the term is deemed satisfied if the associated operand operand2 is not, and vice versa.

operand2 is associated to qaHasToBeResponsible4anyProduct, which is an instance of hasToBeResponsibleForAny. The type that is guarded by this specialization of typeConstraint is identified by the value of constrainedTypeName: 'product'. The constraint instance qaHasToBeResponsible4anyProduct is satisfied, if the element targeted by the association constrainedElement has any relation to any product.

In effect, the term described above has the following informal semantics: The model is deemed unsatisfied, if the root term is not satisfied. The root term is satisfied, if the projectManager is responsible for the projectManual AND exactly one of the following two constraints is satisfied:

1. qaManager is responsible for qaManual
2. qaManager is not related to any product via any specialization of relation.

As described in CHAPTER 8, subterm1 ensures that if qaManager is related to any product, then a responsibility relation to qaManual must exist, too.
Until now, the attribute invalMode that is common to all constraints was ignored. As stated in SECTION 9.1, a term is to be interpreted as not being fulfilled, if any constraint does not point to any target and the value of invalMode is set to invalid. So we may conclude that if in any variant, the projectManual,[1] would be removed, either by change operation or by model reduction after feature configuration, then the rootTerm mainManagementProducts would have to be interpreted as being not satisfied.
This would end up in any such variant being interpreted as an invalid model variant of the original model. In our exemplary DML architecture and model example, this scenario cannot take place, because it happens that there is no change operation able to remove a product from the model, nor can the feature projManagement be configured out. In such a case, i.e. when a constrainedElement cannot be removed from the model in any model variant, then the value assigned to invalMode does not matter and is not interpreted.
On the other hand, qaHasToBeResponsible4qaManual has an invalMode value of false. This means that if and only if the qaManual would be removed from the model by configuration or modification, then the constraint had to be interpreted as unsatisfied. In effect, this allows model elements that are under constraints to be removed from the model without disabling the term containing such constraints.

[1] projectManual is the constrainedProduct of pmHasToBeResponsible4pmManual.

10. Implementation

Contents

This chapter contains a description of an implementation of the concepts that were introduced in PART II. As stated in CHAPTER 1.3, this implementation is primarily an evaluation approach for the concept described in the previous chapters in respect to soundness, consistency, and correctness. In addition it represents an example used to show how the concept can be implemented.
The presented implementation of the DML environment is used as an example for the application of the DML concept. It is provided with a prepared UML model that is consistent with the UML diagrams in CHAPTER 9. It contains selection of features, a downstream model reduction process, transformation according to information provided by change operations, and the transformation of the modified UML models to a logic programming language that is used to reason over constraints located within the model.
The input to this DML environment has to be provided as UML model. As we will see below, selection information is placed along with the whole DML model, i.e. it has to be modeled within the UML model, too.
In addition to implementation issues, this chapter provides a clarification of the exact semantics of the mechanisms introduced in previous chapters.
The chosen strategy was to create a stringent and consistent process going from the beginning, i.e. the definition of a DML framework, to the end, i.e. a development model ready to be used in a project.

	Process step	Implementation	Result format	Section
I.	Selection	Addition of deselection information to the UML model.	UML model	10.2
II.	Reduction	XSL transformation of the model in UML format, regarding the deselection information. Deselection information is discarded.	UML model	10.2
III.	Transformation	Multiple XSL transformations of the reduced UML model. Every instance of change operation is interpreted and executed. Change operations are discarded.	UML model	10.3
IV.	Constraint checking	XSL transformation of the previous UML result model to the formal language Nivel. The model is then tested for constraint inconsistencies.	Nivel model	10.4

Table 10.1: Overview of the DML environment process step implementation

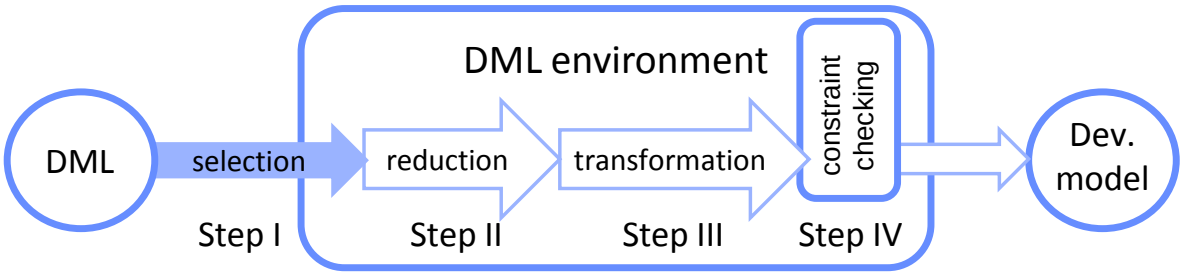

Figure 10.1: Relating TABLE 10.1 to FIGURE 7.7

An important aspect of this strategy is the usage of UML as a modeling language to be able to make this thesis' subjects computer interpretable. I.e., the DML framework, as well as the derived DML architectures and DML's described in CHAPTERS 7 and 9 are modeled as UML models. These models are made persistent as XMI files adhering to the UML XMI 2.1 format, as it is implemented in the export functionality of MagicDraw 16.8.[1] Specifics of the UML modeling of the DML concept are discussed in SECTION 10.1.

The functionality of the DML environment as described in CHAPTER 7.6 is realized as it is shown in TABLE 10.1. A graphical description of the process was initially shown in FIGURE 7.7. This figure was extended with a mapping to the table. The mapping is shown in FIGURE 10.1.

The first step, covered in SECTION 10.2, is the selection of features of a DML. This selection is made manually by a DML manager. The information is stored along with the model information.

[1] `http://www.magicdraw.com/`

Step II concerns the interpretation of this selection information. Any model element, that does not belong to a selected feature is removed from the model. For this transformation, an XSLT interpreter is used: Saxon 9-1-0-7j.[2] Step II is covered in SECTION 10.2, too.
Transformation, i.e. step III, covers the execution of change operations. In the implementation described in this thesis, the semantics of change operations is specified using XSL scripts. Each change operation is executed in order of appearance, finally creating a modified DML. The implementation strategy for step III is described in SECTION 10.3.
In step IV, the DML environment is checking for adherence to constraints contained within the model. In order to make this check possible, the UML model is transformed to a formal language representation of the model. The language used for this purpose is named Nivel. This topic is described in detail in SECTION 10.4.

10.1. DML modeling using UML

In CHAPTER 7, UML representations were provided for the proposed DML framework, an exemplary DML architecture and an exemplary DML. These models were enhanced in CHAPTER 9 by a constraint meta-model. As the figures concerning constraint restricted DML's are more complete, they should be used as a reference when following the implementation details described in this chapter.
In particular, the DML framework was shown in FIGURE 9.1, a DML architecture was presented in FIGURE 9.2, and a DML was illustrated in FIGURE 9.3.
The figures adhere to the graphical UML 2.3 notation [OMG10b]. The models were created using MagicDraw 16.8.[3]
There is an aspect of the models that is not shown in the figures. On page 84, it was stated that for some classes, the attribute 'isLeaf' is set to true. This applies to featureModel, feature, subfeature, term, rootTerm, and constraintOperand. The effect is that those classes must not be specialized in the DML architecture. There is no graphical representation for 'isLeaf' defined by UML, so this aspect has textually been added in the description text covering FIGURES 7.2 and 9.1.
The UML models will later be translated into Nivel [Asi07], another modeling language (see SECTION 10.4.4). There are modeling conventions that must be adhered to in order to allow this transformation.
All elements that belong to the DML framework, have to be placed together in a package named 'framework'. All architecture related elements must be grouped in a package named 'architecture'. The instances of those elements reside in a package 'dml' and in a package 'extension' for the exemplary extension model that was shown in FIGURE 7.5, respectively.

[2] http://saxon.sourceforge.net/
[3] Note that the figures were redrawn manually afterwards for better readability.

Figure 10.2: Meta-model for the definition of selection information

There are more considerations. TABLE 10.2 lists these aspects that have to be regarded when modeling DML's with UML when intending to input this model to the DML environment described in this thesis.
The left column lists the UML entities that may be used in the three layers. The right column lists additional restrictions.

10.2. Configuration: selection and reduction

The chosen strategy used to intertwine a DML model with configuration information was to create an additional meta-model element *featureDeselector* at the same level as the DML framework. FIGURE 10.2 shows how this element is related to the DML framework. featureDeselector is used to model configuration information to be used during reduction of the model.

Selection. A featureDeselector is associated to one or more features via the association *deselects*. In order to define configuration information, instances of featureDeselector have to be added to the DML model. They must then be related to features that are intended to be removed from the model.
The implementation described in this thesis does not check if features are mandatory or optional. I.e., the DML manager is responsible to ensure that only those features are deselected, that are not mandatory in any subfeature context.

Reduction. The integration after selection in the case of a DML is a simple model reduction. After selection, a DML model with featureDeselector instances is interpreted by an XSL script for DML reduction. In effect, the script reduce.xsl copies the whole input model to an output model, stripping it from:

- all featureDeselector instances,
- all feature instances that were referenced by any deselects relation,
- all other model element instances that do not have a belongs_to relation to any of the remaining features, and
- all association instances with outgoing references to any of the removed elements.

Supported UML elements	Restrictions to be considered
DML framework	
• classes • associations • association classes • enumerations • generalization • attributes (String and enumerations)	• Package must be named 'framework'. • All associations must have multiplicities on both ends. • All association ends must be named. • Attributes always have the multiplicity 1.
DML architecture	
• classes • associations • generalization, in case the general is located within the DML framework	• Package must be named 'architecture'. • All classes and associations must specialize an element of the framework. • Associations defined in the framework are implicitly inherited: i.e. no specialization needs to be explicitly modeled, but they may be. • Associations must be specializations. • All associations must have multiplicities on both ends. • All association ends must be named. • The owner of both association ends in an association must be the association itself. • A specialized class or association must have a different name as the father. • The association ends must use the same name as the father association.
DML	
• instance specifications	• Package must be named 'dml' or 'extension'. • Instances may be named, but need not. • All classes of the architecture can be instantiated. • Those classes of the framework can be instantiated that do have the attribute 'isLeaf' set. • Associations must be created explicitly as instances: it does not suffice to create a link between two classes. • In an instance, all attributes defined in the architecture must be assigned a value. • Instances of associations must have a value for both association ends. • Creating diagrammatic links between a class instance and an association instance is optional.

Table 10.2: Considerations for UML modeling of DML's

The script is documented in detail in the appendix on page 153. To understand the script, one needs knowledge of XSLT and the UML XMI format.

10.3. Transformation: execution of change operations

After the reduction of the DML model, the model is transformed. When this model transformation is performed by the DML environment, all instances of c̄hangeOperation are executed. During execution, the semantics of each change operation are interpreted and operationalized. Semantics for a change operation is provided by the DML architecture along with a concrete element specializing c̄hangeOperation.
Transformation in the implementation that is subject to this chapter is realized by the interpretation of XSL scripts. The complete transformation is realized by the repeated execution of the following XSL transformations:

1. Generate an XSL script that is used to be executed in the next step. This XSL script covers the interpretation of exactly one c̄hangeOperation instance residing in the DML model. Such a script is denoted as *effect script*. The result of the script is another XSL script that has to be interpreted by an XSL interpreter.
2. Execute the effect script. It takes a DML model as input. The effect script does not only perform the intended change. It also removes the c̄hangeOperation instance that was the reason for creation of the effect script. Output of the script is a DML model after the intended change.

These two steps are executed repeatedly on the DML model, until no change operation is left in the model. The final state of this DML model is then the transformed development model.
The implementation of this process is described in detail in the appendix on pages 156 ff.

10.4. Constraint checking

In CHAPTER 4, it was motivated to restrict the configuration and modification of DML's to a valid variant space. CHAPTER 9 then introduced a meta-model enhancement for the specification of constraints that can be related to model elements, in order to define which model occurrences are to be regarded as valid, and which not. As an evaluation framework, a logic programming language was chosen. The preliminary strategy is to convert a development model that was expressed in UML and processed so far by the DML environment into a language with stable model semantics [GL88]. The formal basis for this approach is explained below. The language that has been chosen in this thesis was enhanced by ASIKAINEN and MANNISTÖ with

semantics for the formal inspection of models that are oriented on UML [AM09]. The resulting (meta-)modeling[4] language is named *Nivel*.
Nivel has a native support for a UML-like modeling paradigm and is thus capable to both represent DML models, and to be used for automated reasoning over these models.
After translation of a development model from UML to Nivel syntax, the language interpreter decides if all $\bar{r}$ootTerms residing in the model are satisfied. If this test is successful, we know whether the development model that was investigated by this formal test is a valid model, or not. If it is valid, the development model that was output of the change operation transformation step can be regarded as the output of the DML environment. This model is then the resulting development model that can be used by the DML manager for the initialization and execution of projects. This can be supported by further tools that are able to interpret the development model.
Tools like that can provide process enactment or generation of a development model documentation. Nevertheless, such tools are out of the focus of this thesis. But it is important to state that the final result of the DML environment is either the information that the configured variant of the input DML is invalid. Or, the result is the very DML model in UML XMI format that was created in the transformation step, in case of this result being declared as valid by the constraint checking step.

10.4.1. Formal basis

In this thesis, a (meta-)modeling language with a formal semantics is used as a formalization method for the DML concept. The language is named *Nivel*. Nivel's modeling concepts are basically oriented on UML and provide a core set of properties like class, attribute, value, association, generalization, and instantiation [AM09, p. 522]. The language realizes strict meta-modeling [AK02] and incorporates extensions like deep instantiation [AK01].[5]
While usage of the deep instantiation concept is necessary when applying Nivel, the theoretical implications of deep instantiation are out of the scope of this thesis. Nevertheless, the deep instantiation concept brings a benefit when integrated into models of the DML concept. As strict meta-modeling allows a strict detachment of different levels within the model, deep instantiation guarantees the correct assignment of subordinate classes to the right levels. Since the DML concept relies on a strict differentiation between DML framework, DML architecture and DML, usage of three separate levels with deep instantiation appears reasonable.
The language Nivel builds upon stable model semantics for normal logic programs [GL88]. This declarative approach as a basis for Nivel has been chosen in order to provide a formal semantics for Nivel [AM09, p. 522].

[4] The 'meta-' is placed in parentheses, because the language can be used for both defining models and the definition of meta-models for such models.
[5] Strict meta-modeling and deep instantiation will be explained in SECTION 10.4.3.

In logic programs with stable model semantics, rules can be used to formulate constraints over answer sets [Nie99, p. 242]. A solution in an answer set is denoted as a stable model. With stable model semantics, it is possible to calculate an answer set for a model by providing a constraint program, where constraints are expressed as rules [NSS00]. The result is a family of stable models, which all pass the given constraints [MT99, p. 375].
In summary, the underlying idea is to formulate a problem by using logic program rules in a way that the solutions to the problem are covered by the stable models of the rules. This approach differs vastly from the usual logic programming paradigm, which is commonly using a query evaluation with the goal to compute a yes/no answer for a concrete query.
In our application scenario of the DML concept, the answer set paradigm is primarily neglected in respect to the purpose of the constraint restriction mechanism. I.e., in effect, the stable model semantics is used to find exactly a yes/no answer which decides whether a configured and modified DML is within an admissible variant space.
However, in a next step, the whole functionality of the stable model semantics will be integrated and used as a means to create a list of several valid variants. This will be explained in SECTION 10.4.2.
Actually, Nivel is based on an extended stable model semantics called Weight Constraint Rule Language (WCRL) [NSS99, SNS02]. WCRL is extending the stable model semantics with weights that can be applied to rules, which gives more control over the exact amount of predicates that have to be fulfilled in order to infer a result.
Nivel was developed by ASIKAINEN and MANNISTÖ as a formalized meta-modeling language in the environment of several variability languages [Asi08]. One of these languages, namely Forfamel [AMS06], has been elaborated using Nivel [AM09, p. 539]. This meta-model for feature models has been integrated in the DML implementation.
An execution environment for stable model semantics is realized by the program *smodels* and the command line based front-end *lparse*.[6] *lparse* creates ground (variable-free) programs from a Prolog style input syntax and then transforms the result into primitive rules [Nie99, p. 267]. These rules can be given to *smodels* to calculate stable models.
Programs are composed of atoms and inference rules. A problem is formulated using such a program. An atom represents a claim about the universe of discourse. It may be either true or false. An inference rule is used to represent relationships between atoms. A stable model is an answer to a problem, provided as a set of atoms. Constants are always lower cased, variables are upper cased.
See LISTING 10.1 for an exemplary problem: Knights and Knaves.[7] The problem description is:

[6] *smodels* and *lparse* can be obtained at: `http://www.tcs.hut.fi/Software/smodels/`
[7] The example was originally published by SMULLYAN [Smu00]. It was translated to the presented syntax by Tommi SYRJÄNEN and has been provided with *lparse*.

```
1  % There are three persons in this puzzle
2  person(a; b; c).
3
4  % Each person is either knight or knave, but not both.
5  1 { knight(P), knave(P) } 1 :- person(P).
6
7  %% Hint 1: %%
8  % If A tells the truth, both B and C are knights
9  2 { knight(b), knight(c) } 2 :- knight(a).
10
11 % If A lies, it is not possible that they are both knights
12   :- knave(a), knight(b), knight(c).
13
14 %% Hint 2: %%
15 % If B tells the truth, A is a knave and C is a knight
16 2 { knave(a), knight(c) } 2 :- knight(b).
17
18 % If B lies, it is not possible that A is knave and C is knight
19   :- knave(b), knave(a), knight(c).
```

Listing 10.1: Knights and Knaves

> The Island of Knights and Knaves has two types of inhabitants: knights, who always tell the truth, and knaves, who always lie. One day, three inhabitants (*a*, *b*, and *c*) of the island met a foreign tourist and gave the following information about themselves:
>
> 1. *a* said that *b* and *c* are both knights.
>
> 2. *b* said that *a* is a knave and *c* is a knight.
>
> *What types are a, b, and c?*

There is exactly one solution to the problem. The problem will now be formulated so that the result is a stable model.
The first atom in line 2 of the listing states that *a*, *b*, and *c* are persons.[8]
The requirement that a person is either a knight or a knave is formulated in line 5. This is a choice rule which is satisfied when exactly one literal is satisfied, not more or less. This effectively reduces the amount of stable models to those containing only combinations where a person is either marked as a knight or a knave, never more or less.
Now there remain two hints. The first one leads to the knowledge that when *a* is a knight, then both *b* and *c* must be knights, too, since *a* would tell the truth. This is formulated in line 9. The reverse is that if *a* is a knave, then *b* and *c* cannot be knights at the same time. Line 12 contains this condition. A term starting with ':–' disqualifies all models where the right side is satisfied.

[8] The term person(a; b; c) is grounded by *lparse* to three atoms: person(a). person(b). person(c).

The second hint leads to lines 16 and 19. They are created analogously to those from hint 1.
With this problem description, execution of *lparse* and piping the result into *smodels* leads to the following result:

```
Answer: 1
Stable Model: knave(a) knave(c) knave(b) person(c) person(b) person(a)
```

This means that answer 1 is a stable model with all persons being a knave. It is the only stable model, so we know now that the model assumed in the example finally represents a world of liars.

10.4.2. Spanning an array of stable models

The difference in stable model semantics to the usual logic programming paradigm is that is does not only identify a yes/no answer when evaluating a model in respect to the provided constraints. Stable model semantics provides more, as it determines an answer set that may consist of more than one stable model. The mechanism used to realize this are constraint literals like the following:

```
switch(a).
1 { active(N), inactive(N) } 1 :- switch(N).
```

There is an atom *a* that is stated to be a switch. Note that the 'switch' does not hold any semantics, it could be 'foo' or any other name, as well. The rule below the fact statement is that for all atoms N were switch(N) is set as a fact, one of two possible facts is derived: either active(N), or inactive(N). The constraint literal on the left side can be read like: 'The amount of true predicates of those defined within the braces must be at least 1 (left side), and at most 1 (right side)'.
smodels will test all possibilities and output all models that do not collide with any further rules provided in the model. If there are no further rules, the result will be two stable models:

```
Answer: 1
Stable Model: active(a) switch(a)
Answer: 2
Stable Model: inactive(a) switch(a)
```

This mechanism is used below to calculate stable models that satisfy given constraint restrictions provided within a DML.

10.4.3. Nivel

This section describes first the concepts of strict meta-modeling and deep instantiation, since these are incorporated into *Nivel*. Then, Nivel itself is explained, first by explaining its abstract syntax for a better understanding how the language relates to modeling in general, then the concrete syntax is described. Afterwards, some extensions and modifications that had to be implemented into Nivel are explained.

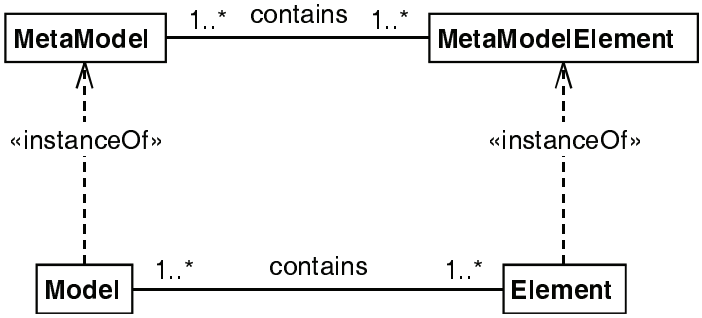

Figure 10.3: Strict meta-modeling [AK02, Fig. 2]

These contain shortcuts for a more convenient modeling process, as well as minor adaptations that had to be made in order to make the DML concept applicable with Nivel.

Strict meta-modeling

Strict meta-modeling guarantees that models concerning different meta-levels have well-defined boundaries. In particular, this modeling paradigm interprets the instance-of relationship at the granularity of individual model elements [AK02, p. 9], see also FIGURE 10.3. This is framed by the following rule:

> "*In an n-level modeling architecture,* M_0*,* M_1 *...* M_{n-1}*, every element of an* M_m*-level model must be an instance-of exactly one element of an* M_{m+1}*-level model, for all* $m < n - 1$*, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)=level(Y).*"
> [AK02, p. 9]

In effect, this rule guarantees that no element is an instance of an element residing two or more levels above. In addition, single classification is enforced, so that an element cannot be an instance of two elements in a meta-model.

Deep instantiation

A consequent realization of strict meta-modeling is resumed with the concept of *deep instantiation* [AK01]. The concept refrains from the case of *shallow instantiation*, where a "two-levels only" modeling philosophy is present with only a model level and a meta-model level [AK01, p. 27].
In a model environment with several meta-model levels, an element can have both class and object properties. The occurrence of this phenomenon is apparent in practice and has been investigated [Ode94, Atk97]. ATKINSON denoted an element with these properties a *clabject* [Atk97, p. 96].
Deep instantiation is an attempt to address this phenomenon with an explicit instantiation model. A goal of this approach is to formulate a mechanism "in

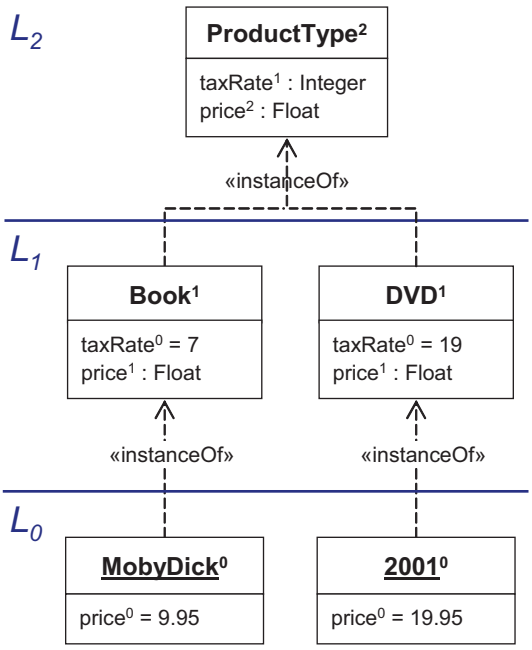

Figure 10.4: Deep instantiation [KS07, Fig. 6]

which a modeling element's class features can be acquired automatically by the instantiation step rather than always having to be defined explicitly" [AK01, p. 28].

The main aspects used to model a deep instantiation structure are the concepts of *level* and *potency*. Both are integer values that have to be clearly defined for every model element.

The level of an element indicates the meta-level the element is located at. An element at level 0 is at the lowest level. In reference to the MOF meta-levels, it's an M_0 element [OMG06]. Above the level of an element, another meta-level can be present, which has a level that is higher by 1.

The potency is slightly different. It indicates the amount of instantiations of an element that can be made. Each instance inherits the potency of its meta-model element, reduced by 1. I.e., an element at level 2 with a potency of 1 is a meta-model element for possibly created instances at level 1. These instances cannot be further instantiated, as they have a potency of 0.

An element with a potency of 2 can be instantiated twice: first, to an element one level below, with a potency of 1. Then, this element can be instantiated once to an instance at another level below, which will have a potency of 0. Such an instance cannot be further instantiated.

Note that the potency concept is largely, but not completely independent from the level of an element. The potency must never be higher than the level an element resides on, since an element at level 0 with a potency > 0 could not be instantiated.

FIGURE 10.4 shows an example of a model with deep instantiation. The potency is noted like a mathematical potency beside a class or property name. As we can

see at level L_2, there is a class *ProductType* with a potency of 2. It has a property *taxRate* with a potency of 1, indicating that the lowest instantiation of this property must be one level above that of ProductType. One level below, at L_1, we can see instances of ProductType: *Book* and *DVD*. They have a potency of 1, since every instantiation reduces the potency of the instance by 1. taxRate is instantiated, too, but as it reaches a potency of 0, it represents the actual value used for taxRate. It indicates that each Book has a taxRate of 7 and each DVD a taxRate of 19.
As the example shows, the deep instantiation concept allows to model values that are applicable to a complete class of objects. *MobyDick*, being an L_0 instance with a potency of 0, has its own attribute values. But a property like taxRate ought to be stated for all elements that are instances of the Book class, not for each Book instance separately. Defining the potency of taxRate at the highest level allows to explicitly model such a design decision.

Nivel's abstract syntax

Nivel is primarily a meta-modeling language. Its abstract syntax represents a reduced version of UML. It shares the notions of class, attribute, association, cardinality, and generalization.
FIGURE 10.5 shows the abstract syntax as it was summarized by ASIKAINEN and MANNISTÖ. A detailed description of the language is provided in [AM09].
We will take a quick look at the elements that are used later on for the formalization of the constraint mechanism.
The main elements of those depicted by FIGURE 10.5 are Class, Association, Attribute, Value, Role, as well as the instanceOf and the subclassOf relationships.
A Class can be an instance of another Class. In this case, the instance is located at a level lower by 1. Its potency is reduced by 1, too.[9]
A Class may be a top-level Class. This indicates that it is not an instance of another Class and no Class is located at a higher level than this Class.
Associations are related to Role and Class via a ternary relation playsRoleIn. An Association connects Classes to each other that play a certain role in an Association.
A Class may be related to Attributes via hasAttr. If an Attribute has been instantiated until its potency has reached 0, a Value must be assigned to the class via hasValue.

Nivel's concrete syntax

TABLE 10.3 lists all Nivel syntax elements that are used in this thesis. The exact semantics of the predicates were described by ASIKAINEN and MANNISTÖ [AM09]. We will not discuss them in detail here, but we need an overview on what is used later on during implementation. The definition of Nivel's rules was published on the web page of the SoberIT Software Business and Engineering Institute [Asi07].[10]

[9] See above in the explanation of deep instantiation for the notions of level and potency.
[10] `http://www.soberit.tkk.fi/nivel/`

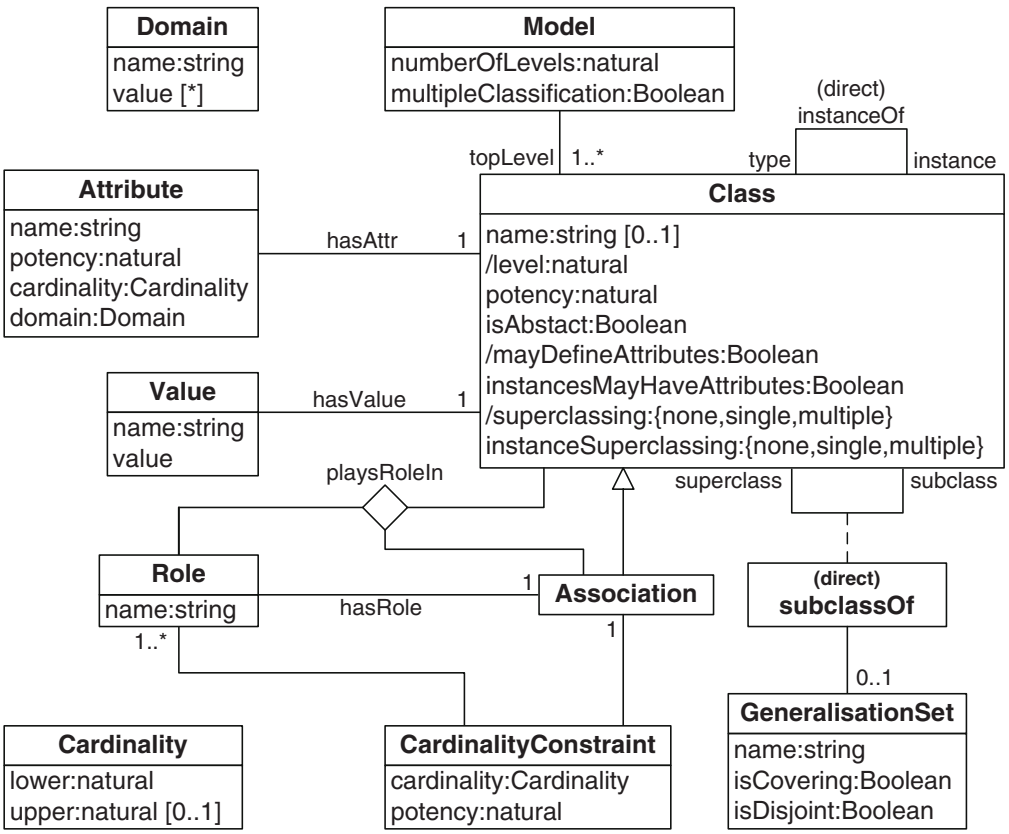

Figure 10.5: Abstract syntax of Nivel [AM09, Fig. 5]

Note that the *lparse* interpreter identifies any string from the occurrence of '%' on as a comment.
An important aspect is that most predicates have different modes to be used in. The normal mode is enabled by using the predicate name without any appendices. In this case, the predicate represents is an *actual* fact. In contrast, a predicate may represent a *possible* fact.
The distinction between actual and possible facts is made in Nivel to span the field for the determination of stable models. An input Nivel model interpreted by *smodels* must have possible facts so that *smodels* can calculate all stable models that satisfy all given rules. The output model then only contains actual facts that were determined during the calculation process as being models that do not break with any rules.
For example, a class may be actual (**class**(c)), or it may be possible (**class_p**(c)). If a class is stated to be actual, *smodels* must interpret this class as actually being present in all valid model variants it goes through. If it is only possible, *smodels* is free to drop the idea of having this class in a stable model. This effectively enlarges the space of admissible stable models.
Modes are applied with an underscore and a following letter sequence. Available modes are actual (no appendix), possible (p), direct (d), transitive possible (tp), and declared (D).[11]
Actual and possible facts were described above. A direct fact implies an actual relationship between two model levels that are adjacent to each other. In particular, this means that the statement of a direct fact makes it an actual fact. For example, a direct instanceOf relationship is present between the classes c̄ontentElement and prod̄uct in the DML architecture: prod̄uct is a direct instance of c̄ontentElement, since there are no other instanceOf relationships between them.
A transitive fact may span more than one level. projectManual̄ is a transitive instanceOf c̄ontentElement. This relationship is spanned over 2 levels via prod̄uct.
Note that direct relationships imply actual relationships, while transitive relationships don't. Transitive relationships are always only related to possibility, meaning that is may become actual, but need not.
Declared facts imply both possibility and directness, while the latter is only used if applicable. I.e., if a declared fact is provided, *smodels* is free to actualize the fact, or not.
The different modes will now be subsumed in reference to TABLE 10.3 for the instanceOf predicate. The predicate comes with four variants: actual (_), possible (p), direct (d), and declared (D). An actual instanceOf relation must be present in the stable model calculated by *smodels*. A possible instanceOf relation may be present, but needs not. Direct instanceOf implies an actual relationship that does span over exactly one level boundary. A declared instanceOf relation is a possible relation that may either be a direct relationship, or dropped from the stable model. A possible

[11] These are only the modes that were used for the implementation described in this thesis. The complete list of modes is described in [AM09].

Predicate	Variants	Semantics
class(c)	_ *p*	Object constant *c* represents a class
topLevel(c)	_ *D*	Class *c* is on top level
subclassOf(a,b)	_ *p d D*	Class *a* is a subclass of class *b*
instanceOf(i,t)	_ *p d D*	Class *i* is an instance of class *t*
instanceOf(i,t,o)	*tp*	Class *i* is an instance of class *t* of order *o*
hasPotency(c,p)	_ *D*	Class *c* is of potency *p*
hasAttr(c,n,p,d,l,u)	_ *p D*	Class *c* has attribute named *n* with potency *p*, domain *d*, cardinality lower bound *l* and upper bound *u*
hasValue(c,n,v)	_ *p D*	Class *c* has value *v* under name *n*
association(a)		Class *a* is an association
hasRole(a,r)	_ *p D*	Association *a* has role *r*
playsRoleIn(c,r,a)	_ *d p D*	Class *c* plays role *r* in association *a*

Symbols in the *Variants* column: '*p*' possible, '*d*' direct, '*D*' declared, '*t*' transitive, '_' actual. In case more than one variant is defined, the semantics are described for the actual variant

Table 10.3: Nivel's predicates, reduced list [AM09, Tab. 1]

instanceOf relation may be transitive, i.e. it may span other instanceOf relations on the way from the classifier to the instance.

Adaptations to Nivel's syntax

Two kinds of adaptations were implemented for the usage with Nivel. There have been minor adaptations in Nivel's logic to be able to model the DML concept, and some shortcuts were added for a more convenient usage of the modeling language.

Adaptations in the logic. In the original Nivel rule set, the following rule was defined [Asi07]:[12]

```
360 % Only top-level classes may declared roles
361 % (in appendix A)
362 error(falseEndDeclaration) :- hasRole_D(A,R), not topLevel_D(A).
```

This rule identifies all classes that are not located at the uppermost level, but have new role declarations. Such classes are claimed to be an indication for an erroneous model. In the DML framework, r̄elation is a subclass of c̄ontentElement. I.e., r̄elation cannot be a top level element. Thus, the role above would disqualify any model with an association being a subclass and defining its own roles. For this reason, the line above has been commented out for usage of the Nivel framework with the DML concept.

[12] The line numbers refer to the lines in the original nixel.txt linked in [Asi07].

```
1 association(owns).
2 hasPotency_D(owns,2).
3 hasRole_D(owns,owner).
4 hasRole_D(owns,owned).
5 playsRoleIn_D(person,owner,owns).
6 playsRoleIn_D(item,owned,owns).
```

Listing 10.2: Creating an association with nivel

Shortcuts. When using the Nivel language, a lot of complex language patterns appear again and again, with minor variations. In order to enable a more concise syntax building, some abbreviations have been added to Nivel. This enables convenience in modeling Nivel models, and it provides a better overview of what has actually been modeled.
For example, modeling an association with Nivel is actually more than the simple usage of the *association* predicate. We have to define association ends (roles) and what classes are attached to these ends, as well. Such a definition may look like the syntax shown in LISTING 10.2.[13]
The listing contains a list of atoms. These atoms represent an association named *owns*, which relates an *item* to a *person*. First, the association must be created and it gets a potency. Then, in lines 3 and 4, the association ends *owner* and *owned* are created. The last two lines relate these roles to *person* and *item*, respectively.
In order to be able to easily create associations of that kind, a new predicate has been introduced: *scn_association_D*. It allows the definition of all the information provided in LISTING 10.2 with a single line:

```
scn_association_D(owns,2,owner,person,owned,item).
```

The first parameter is the name the association has to get. The second indicates the potency, while the next parameters appear pairwise. Parameter 3 and 4 represent one association end, where the first defines the name of the role, and the second defines the class to which the role belongs. The last two parameters are analogously.
LISTING 10.3 shows how the predicate *scn_association_D* is embedded into Nivel. Every time this predicate is used, it implies the association-related predicates, effectively creating an association just like writing the whole code manually. Note that the parameters there are variable names, not constants.
Another shortcut is the creation of a sub-association via *scn_subAssociation_D*. The parameter list is similar to that of *scn_association_D*. The only difference is that no potency is needed, because it can be determined from the parent's potency. Instead, the parent must be provided, as the second parameter *Superclass*:

```
scn_subAssociation_D(Name,Superclass,Role1,Type1,Role2,Type2)
```

Since a sub-association needs less information to be created, LISTING 10.4 showing the shortcut code is much shorter than LISTING 10.3.

[13] Note that all predicates that are specific to Nivel, as well as shortcuts, are presented in a bold style.

```
1  topLevel_D(Name) :-
2      scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
3  association(Name) :-
4      scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
5  hasPotency_D(Name, Potency) :-
6      scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
7  hasRole_D(Name, Role1) :-
8      scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
9  hasRole_D(Name, Role2) :-
10     scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
11 playsRoleIn_D(Type1, Role1, Name) :-
12     scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
13 playsRoleIn_D(Type2, Role2, Name) :-
14     scn_association_D(Name, Potency, Role1, Type1, Role2, Type2).
```

Listing 10.3: Nivel shortcut (scn) for the creation of an association

```
1  instanceOf_D(Name, Superclass) :-
2      scn_subAssociation_D(Name, Superclass, Role1, Type1, Role2, Type2).
3  playsRoleIn_D(Type1, Role1, Name) :-
4      scn_subAssociation_D(Name, Superclass, Role1, Type1, Role2, Type2).
5  playsRoleIn_D(Type2, Role2, Name) :-
6      scn_subAssociation_D(Name, Superclass, Role1, Type1, Role2, Type2).
```

Listing 10.4: Nivel shortcut (scn) for the creation of a sub-association

Another shortcut is *scn_subAssociationMultiple_D*. It is used in a context when a class with outgoing associations that is residing on an upper level is subclassed to a lower level. All associations of the parent class have to be transferred down to the subclass, as well. The predicate is modeled in LISTING 10.5. The parameter list resembles the list for *scn_subAssociation_D*, as well as the handling within LISTING 10.5, but there is a significant difference in the usage of the last two parameters. These are not used to define a role playing (*playsRoleIn_D*), but to define a set of role playings. For each subclass of the type provided with *TypeRole2*, a role playing is defined. This ensures that a subclass of an association is able to be associated to all subclasses of those classes the parent association can be associated with.

```
1  instanceOf_D(Name, Superclass) :- scn_subAssociationMultiple_D(Name,
       Superclass, Role1, TypeRole1, Role2, TypeRole2).
2  playsRoleIn_D(TypeRole1, Role1, Name) :- scn_subAssociationMultiple_D(Name,
       Superclass, Role1, TypeRole1, Role2, TypeRole2).
3  playsRoleIn_D(SubInstance, Role2, Name) :- scn_subAssociationMultiple_D(Name
       , Superclass, Role1, TypeRole1, Role2, TypeRole2),
4      instanceOf_p(SubInstance, TypeRole2).
```

Listing 10.5: Nivel shortcut (scn) for the creation of multiple sub-association relations

10.4.4. DML expressed in Nivel

As stated in SECTION 10.1, UML modeling was used to describe the structure of the DML framework, an exemplary DML architecture, and an exemplary DML. In order to use Nivel as logic programming language to define these structures, a mapping is needed for the creation of Nivel models from UML models. This section provides such a mapping. The mapping has been verified by implementation of an automated transformation for this purpose.
The models shown in CHAPTER 7 were modeled using MagicDraw and then exported to the UML 2.1 XMI format provided by a MagicDraw internal exporter [OMG07a]. The XMI file is looped through an XSL transformation that creates a Nivel syntax as a result [W3C99].
Three transformations are needed to create a Nivel model that can be run through *smodels*. One for the creation of the syntax for the DML framework, one for a concrete DML architecture, and one for a concrete DML.
A word on the level structure in both UML and Nivel models. As shown by FIGURE 5.1, the DML architecture is created using the DML framework by specialization. Then, a DML is created by instantiation, using the DML architecture. In the context of Nivel, it has been chosen to map this to the process of instantiation in both cases. This is due to the benefit that usage of instantiation in an environment supporting deep instantiation allows a strict and definite separation of DML framework, DML architecture, and DML. DML framework elements are always on level 3, elements of the DML architecture on level 2, and DML element reside on level 1.
This leaves level 0 for actual projects where a DML is used as a meta-model. However, this aspect is not regarded in this thesis.

DML framework

The DML framework is transformed to Nivel model elements by an XSL Transformation (XSLT) [W3C99]. The transformation script is described in detail in the appendix on pages 165 ff. Here, we will take a look at exemplary translations of framework elements into Nivel syntax.
At the top level (level 3), which is the level all elements in the DML framework reside in, a potency of 3 is assigned to each element.
For example, the *modelElement* shown in FIGURE 9.1 is translated to Nivel syntax as:

```
topLevel_D(modelElement).
hasPotency_D(modelElement,3).
```

Nivel needs the information whether an element is at the top level, so all elements in the DML framework that are not subclasses of other elements are declared as *topLevel_D* elements. Nivel infers the *class_p* predicate from all *topLevel_D*, so there is no need to define that explicitly. *modelElement* is declared with a potency of 3.
For definition of associations like *belongs_to*, a shortcut has been provided in SECTION 10.4.3. The syntax for this association is:

```
scn_association_D(belongs_to,3,capsule,feature,element,modelElement).
```

Note that the role names provided to *scn_association_D* were actually shown in FIGURE 9.1. These are the labeled association ends.

Any class that is not a top level element, like contentElement, does not need the definition of a top level predicate, nor a potency. Instead, it is modeled as subclass of another class. For example contentElement, being a subclass of modelElement, is represented by:

```
subclassOf_D(contentElement,modelElement).
```

Any enumeration, like cardinalityType with its two literal values mandatory and optional, are translated into a shallow value representation, using the *contains* predicate. Whenever an attribute makes use of the enumeration as its type, it's referring to the possible enumeration literals as value domain.

For example, the cardinalityType literal values are represented by the following lines.

```
contains(cardinalityType,mandatory).
contains(cardinalityType,optional).
```

A class such as subfeature having an attribute of type cardinalityType is referring to the enumeration value domain, thus allowing assignments of values within the given domain. For subfeature, the attribute cardinality is assigned via:

```
hasAttr_D(subfeature,cardinality,2,cardinalityTypeDomain,1,1).
```

The following lines of Nivel syntax represent an arranged excerpt of the DML framework model that was shown in FIGURE 9.1. It is provided here to show an example of the transformation from UML to Nivel.

```
...
topLevel_D(modelElement).
    hasPotency_D(modelElement,3).

subclassOf_D(contentElement,modelElement).

scn_association_D(relation,3,source,contentElement,target,contentElement).
    subclassOf_D(relation,contentElement).
...
```

We can see the declaration of the framework entity modelElement, along with its specialization contentElement. This contentElement is bound within the association relation both with the association ends source and target. As was shown in FIGURE 9.1, relation is a further specialization of contentElement.

The complete transformed DML framework model is provided for reference in the appendix on pages 177 ff.

DML architecture

The transformation of a DML architecture is described in detail in the appendix on pages 169 ff. Like it was done for the framework, we will here take a look at exemplary transformation results.
All elements in the architecture that are specializing elements in the framework are in Nivel instances of their parent. For example product, which is a specialization of contentElement, is represented in Nivel syntax as an instance of contentElement:

```
instanceOf_D(product,contentElement).
```

All associations are represented by instances as well. In addition, all association ends and their appropriate type are defined as a role playing via *playsRoleIn_D*. For example, the relation responsibility has an association end target. This is related to the type product by the following line:

```
playsRoleIn_D(product,target,responsibility).
```

A specialty of the DML architecture in Nivel is that if there are classes in the DML framework that have to be instantiated in a concrete DML, then we need an instance of these classes on the level of the DML architecture. This is done by the creation of *implicit* classes and associations. This means that for any class in the DML framework, an implicit class is placed within the Nivel syntax representing this class to be ready for instantiation in a DML.
In particular, this involves the creation of artificial classes that were not modeled within the architecture. An example is feature. Since features have to be instantiated in a DML, we need a representative of this class in the DML architecture. This representative is created using the name of the class with an appendix '_ARCH':

```
instanceOf_D(feature_ARCH,feature).
```

To enable such classes to be compatible with associations, we need the definition of *scn_subAssociationMultiple_D* predicates.[14] The usage of a predicate like the following allows a product to have associations to any contentElement, as this property is inherited by the association relation.

```
scn_subAssociationMultiple_D(relation_ARCH,relation,source,product,target,
    contentElement).
```

The following lines of Nivel syntax represent an arranged excerpt of the exemplary DML architecture model that was shown in FIGURE 9.2. It is provided here to show an example of the transformation from UML to Nivel.

```
...
instanceOf_D(product,contentElement).
    scn_subAssociationMultiple_D(relation_ARCH,relation,source,product,
        target,contentElement).
    scn_subAssociationMultiple_D(relation_ARCH,relation,target,product,
        source,contentElement).
```

[14] The purpose of this predicate is described in SECTION 10.4.3.

```
    scn_subAssociationMultiple_D(constrainedElement_ARCH,
        constrainedElement,element,product,constr,constraint).

instanceOf_D(responsibility,relation).
    playsRoleIn_D(role,source,responsibility).
    playsRoleIn_D(product,target,responsibility).
...
```

The code shows the specialization of contentElement to product, and of relation to responsibility. As stated above, specialization in a UML model between the DML framework and the DML architecture is realized in Nivel with instantiation.
Some related associations of contentElement were automatically specialized via the predicate **scn_subAssociationMultiple_D** for usage in a DML for model consistency.
The association ends of relation have to be defined for responsibility, too. The predicate **playsRoleIn_D** is used for this definition. As we can see, the association ends are typed with classes from the DML architecture.
The complete transformed DML architecture model is provided for reference in the appendix on pages 179 ff.

DML

Like the translations described in the previous two subsections, the generation of Nivel syntax from a UML DML model at the instance level is provided in the appendix. The particular script covered in this section is described on pages 172 ff. The main purpose of the script is to go find all instances that reside in a DML and place them in the Nivel model. An instance projectManual of class product will be represented by:

```
instanceOf_D(projectManual,product).
instanceOf_d(projectManual,product).
```

The first line declares the instance, the second line makes sure that it is actualized, meaning that the *smodels* parser is not able to remove the instance from the model to find more stable models.
If an instance has a classifier located at the DML framework, a '_ARCH' is attached to the classifiers name. This is needed to find the correct instance at level 2 of the Nivel model, i.e. the architecture level. The creation of those implicit elements has been described in the previous section.
In the example below, we see an instance of a belongs_to relation. Since the relation has not been named in the DML model, the XSLT interpreter places a name for it: d1e206.

```
instanceOf_D(d1e206,belongs_to_ARCH).
instanceOf_d(d1e206,belongs_to_ARCH).
```

When an attribute is attached to an instance and this attribute represents an enumeration value, the value is stored as a **hasValue_D** predicate. For example, the

constraint instance pmHasToBeResponsible4pmManual has its attribute invalMode set to the value invalid by:

```
hasValue_D(pmHasToBeResponsible4pmManual, invalMode, invalid).
```

If the instance is subject to an association, the role playing is stored with the predicate **playsRoleIn_D**. For example, the projectManual plays the role target in the responsibility pmResp4pm:

```
playsRoleIn_D(projectManual, target, pmResp4pm).
```

The following lines of Nivel syntax represent an arranged excerpt of the exemplary DML model that was shown in FIGURE 9.3. It is provided here to show an example of the transformation from UML to Nivel.

```
...
instanceOf_D(projectManager, role).
instanceOf_d(projectManager, role).

instanceOf_D(projectManual, product).
instanceOf_d(projectManual, product).

instanceOf_D(pmResp4pm, responsibility).
instanceOf_d(pmResp4pm, responsibility).
    playsRoleIn_D(projectManager, source, pmResp4pm).
    playsRoleIn_D(projectManual, target, pmResp4pm).
...
```

We can see here the instance declaration of projectManager, projectManual, and the responsibility relation pmResp4pm between them. The association ends of pmResp4pm are assigned via usage of the predicate **playsRoleIn_D**.
The complete transformed DML model is provided for reference in the appendix on pages 181 ff.

10.4.5. Semantics of features

As described above, the feature model structure in the DML framework has been adapted from Forfamel [AMS06], a feature modeling approach. This approach has been implemented in Nivel by ASIKAINEN and MANNISTÖ [AM09]. This implementation has been contributed on the Internet [Asi07] and included into the DML framework as is, with a slight alteration.

features are denoted with 'featureType' in the Nivel implementation Forfamel, which had to be changed to fit into the DML framework. This was done by renaming 'featureType' to 'feature'.
Another important aspect in respect to the semantics of features is the ability to describe how a model has to be reduced according to configuration information. This part of the semantics has already been covered in SECTION 10.2.

```
1  constraintIsFulfilled(Constraint) :-
2      instanceOf_D(Constraint, hasToBeResponsibleFor),
3      instanceOf_D(Product, product),
4      instanceOf_D(Role, role),
5
6      instanceOf_D(ConstrainedRole, constrainedRole),
7      playsRoleIn_D(Role, element, ConstrainedRole),
8      playsRoleIn_D(Constraint, constr, ConstrainedRole),
9
10     instanceOf_D(ConstrainedProduct, constrainedProduct),
11     playsRoleIn_D(Product, element, ConstrainedProduct),
12     playsRoleIn_D(Constraint, constr, ConstrainedProduct),
13
14     instanceOf_D(Responsibility, responsibility),
15     playsRoleIn_D(Role, source, Responsibility),
16     playsRoleIn_D(Product, target, Responsibility).
```

Listing 10.6: constraintIsFulfilled semantics for hasToBeResponsibleFor

10.4.6. Semantics of constraints

We now get to know a DML architecture-specific predicate: *constraintIsFulfilled*. Remember the exemplary constraint-specialization that was described in CHAPTER 9.2: hasToBeResponsibleFor. During introduction of this element, it was stated that the exact semantics of constraints would later be covered. It will be described now, and in more detail in the appendix on pages 185 ff. See LISTING 10.6 for the semantics of hasToBeResponsibleFor.
An instance of the constraint hasToBeResponsibleFor is satisfied, if the predicate *constraintIsFulfilled* is true for this instance.
As LISTING 10.6 shows, the predicate is evaluated true, if there exist instances *Constraint* of class hasToBeResponsibleFor, *Product* of class product, *Role* of class role, and *Responsibility* of class responsibility with the following properties:

- *Role* is attached to *Constraint* via an instance of association constrainedRole.
- A *Product* instance is attached to *Constraint* via an instance of the association constrainedProduct.
- *Role* is responsible for *Product*. This is modeled by them both being assigned to the same instance *Responsibility* of responsibility.

An example corresponding to FIGURE 9.3 is pmHasToBeResponsible4pmManual. It is an instance of hasToBeResponsibleFor, the very constraint that is defined in LISTING 10.6. The constraint is resolved as true, with a variable assignment stated in TABLE 10.4.
The result is that the predicate constraintIsFulfilled(pmHasToBeResponsible4pmManual) is set.

Variable	Model element assigned to variable
Constraint	pmHasToBeResponsible4pmManual
Product	projectManual
Role	projectManager
Responsibility	pmResp4pm
ConstrainedRole	association of type constrainedRole between projectManager and pmHasToBeResponsible4pmManual
ConstrainedProduct	association of type constrainedProduct between projectManual and pmHasToBeResponsible4pmManual

Table 10.4: Assignment of variables used in LISTING 10.6 that satisfy the constraint pmHasToBeResponsible4pmManual

```
1  constraintIsFulfilled(Constraint) :-
2      instanceOf_D(Constraint, hasToBeRelatedToAny),
3      constrainedType(Constraint, Type),
4      instanceOf_D(TypeInstance, Type),
5      instanceOf_D(Element, contentElement),
6      constrainedElement(Constraint, Element),
7
8      instanceOf_tp(Relation, relation, 2),
9      playsRoleIn_D(Element, source, Relation),
10     playsRoleIn_D(TypeInstance, target, Relation).
```

Listing 10.7: constraintIsFulfilled semantics for hasToBeRelatedToAny

LISTING 10.7 presents another constraintIsFulfilled predicate. This one is related to the hasToBeRelatedToAny architecture element. It is satisfied for an instance of hasToBeRelatedToAny if there is a contentElement of a particular type that is related to the element identified by the association constrainedElement within hasToBeRelatedToAny. The constrained element is identified by a constrainedElement predicate and the corresponding type is specified by the constrainedType predicate. These predicates are formalized in the shortcuts section on pages 185 f.

In the DML example provided in this thesis, the predicate constraintIsFulfilled is fulfilled for the constraint hasToBeRelatedToAny with the variable assignment shown in TABLE 10.5.

At this point, it is important so understand that constraintIsFulfilled predicates are domain-specific. I.e., they are defined during definition of a DML architecture, since this is domain-specific, as well.

A constraint is a conceptual entity that makes use of formal rules. As described in CHAPTER 9.1, there are two kinds of constraints. One of them is represented by constraint, the other by typeConstraint. A typeConstraint is like a constraint but can additionally refer to a class name via its attribute constrainedTypeName.

The exact semantics of the described constraints is described in Nivel syntax in

Variable	Model element assigned to variable
Constraint	qaHasToBeRelated2anyProduct
Type	product
TypeInstance	qaManual
Element	qaManager
Relation	qaResp4qa

Table 10.5: Assignment of variables used in LISTING 10.7 that satisfy the constraint qaHasToBeRelated2anyProduct

the appendix on pages 185 ff. The rules are an implementation of the following requirements:

1. A constraint may either be satisfied or unsatisfied. A satisfied constraint is a constraint where all targeted elements fulfill formal rules defined domain-specifically for a constraint specialization in the DML architecture.
2. For any combination of satisfied and unsatisfied constraints that are organized in terms, the variants are tested. The result may be a set of stable models where given constraints are satisfied that are needed to be satisfied according to the associated term.
3. All rootTerms must be satisfied, otherwise the containing model variant must be deemed invalid.

The implementation of these restrictions is pointed out in the appendix at the appropriate code lines.

Part III.

The Results

Contents of the Third Part

11. Discussion

Contents

As CHAPTER 5 described, the proposed solution concept is a three-leveled approach used to support variability of development models. The targeted variability dimensions are configurability, extensibility, and modifiability. In addition, a constraint mechanism has been added to the DML concept that allows the restriction of variants to a variant space of variants that satisfy a given set of domain-specific rules.

This chapter provides a discussion on the creation process of development models using the DML concept, on the variability mechanisms, and on the constraint restriction mechanisms. In addition, we will take a look at the problems that were identified in CHAPTER 2. Finally, a short view on the chosen and on possible technologies to implement the DML concept is taken.

11.1. Review on the development model creation process

The topic of this section is the DML concept in general. First, we will take a look at some limitations, then at further implications of the concept.

11.1.1. Limitations

In the proposed DML approach, there are some limitations that should be discussed.

Order of change operations. In the presented approach, there is no possibility to define the order of change operations. A concept is needed for the organization engineer to be able to clearly specify an order, in which the operations have to be performed. Otherwise, the resulting development model will be undefined and may vary between different implementations of the DML approach. In CHAPTER 7.6, it was defined that change operations are executed in order of their appearance in the model's persistent file, but this is only a rough approach to operation ordering. It may be necessary to define a process meta-model for change operations to be able to explicitly model operation ordering. But since such a meta-model is not easily defined due to side-effects of configurability[1] and extensibility,[2] operation ordering was kept simple to reduce the concept complexity.

No support for running projects. Upgrading a customized DML after a new release of the adapted standard is done by an organization engineer. Running projects cannot simply be ported to the new meta-model. It is not an ambition of the proposed concept to enable such porting.

Ambiguities in association specialization vs. subsetting and redefinition. Nivel supports specialization of associations by allowing the definition of one association to be a subclass of another. However, the "interaction of association specialization with association end redefinition and subsetting is not defined" in UML 2.1.1 [OMG07b, p. 41], which is the basis for Nivel [AM09, p. 545].
A more recent view on subsetting and redefinition, as it is presented in UML 2.3 implies that "any association end that subsets or redefines another association end forces the association of the subsetting or redefining association end to be a specialization of the association of the subsetted or redefined association end respectively" [OMG10b, p. 40].
Thus, this connection between specialization and subsetting resp. redefinition was not implemented in Nivel. As a consequence, the UML model of an exemplary DML

[1] An operation that is removed from the model because of feature selection may render a given order invalid.
[2] An operation added to a model must be related to the order of pre-existing change operations.

architecture, as shown in FIGURE 9.2, makes use of association specialization and subsetting, but without association end renaming during subsetting. Otherwise, i.e. if association end names in the different generalization layers differed, the resulting Nivel model would be invalid.

Non-trivial specification of semantics. With the approach for implementation presented in CHAPTER 10, it is not an easy task to specify semantics for change operations (XSLT) and constraints (Nivel rules).
These technologies generally don't belong to the technology portfolio a development model engineer is familiar with.

11.1.2. Implications

After having taken a look at the limitations, we will reflect some further implications of the proposed approach.

Extension models represent change lists. An extension model does only contain organization-specific content elements and change operations that alter content elements of the standard. Thus, the extension model can be used as documentation of the adaptation steps and may be transformed to a change list representation. Such a change list is an important asset for future revisions of the development model [Arm08, p. 4].

Write-protection of the reference model. No element of the original model has to be overwritten in order to achieve the intended result. Instead, all information is expressed as additive information in an extension model. This is enabled by the fact that relations between extensible elements are modeled as first-class entities referencing other elements.
As an example, we will take a look at a *role* with an outgoing second-class association *owned_products*.[3] This association is used to mark all *work_products* the *role* is responsible for. In this case, in order to add a *work_product* to the list of owned products, we would be obliged to either change the *role* instance itself by overwriting its association, or we needed a change operation performing that purely additive variability.
With an association class *owned_products_first_class*, we might simply add such a relation element to the extension model and thus add a *work_product* to the list of owned products of a *role* without directly changing the *role* itself.

Tailorability of extensible relations An effect of extensibility the V-Model XT makes use of is the possibility to tailor relations. This was used to adapt responsibility relations to a tailoring profile.

[3] I.e., the association is part of the *role* class and has no separate identifier.

Change operations modify change operations. According to the DML framework, a specialization of c̄hangeOperation may be defined in a DML architecture that can be used to modify c̄hangeOperation instances. It has not been investigated if this possibility is necessary for an increased flexibility of adaptations, or if any issues emerge from this possibility.

Change operations are feature-controlled. The c̄hangeOperation class is a specialization of m̄odelElement. As such, it has a b̄elongs_to association to f̄eature. This includes c̄hangeOperations in the configuration of a DML. Since in the DML environment, change transformation is deferred until after configuration, in effect, c̄hangeOperations are tailorable.
This fact creates the benefit that changes due to c̄hangeOperations can be finely tuned to particular project contexts. On the other hand, a side effect of this flexibility is that for each c̄hangeOperation, it must be pondered and decided to which f̄eatures it b̄elongs_to.

Change operation restriction. Often, an organization adapting a standard development model needs to retain conformity with the standard model. In those cases, any change to the original model must be checked for conformity.
The core asset engineer can restrict the possibilities of the organization engineer by an accurate provision of change operations in the DML architecture. By giving a restricted tool set to the organization engineer, the core asset engineer can, up to a certain degree, constructively guarantee that the adapted process model will conform to the intentions of the development model.[4] However, this guarantee only applies when the organization engineer does not change the reference model. If renaming or deletion of content elements may result in a non-conforming process,[5] the corresponding change operations simply mustn't be inserted into the tool set for adaptation.

11.2. Review on variability mechanisms

This section contains a brief overview of implementation issues concerning the variability mechanisms that were introduced in this thesis: configurability, extensibility, and modifiability.

11.2.1. Configurability

The *feature model* elements that are integrated into the DML framework are the key to configurability. They allow the definition of a graph that can be used for the

[4] See CHAPTER 4.1.
[5] Note that the exact definition of a conforming process is up to the core asset engineer.

configuration of a DML by means of choosing or discarding optional features.

Early configuration. The feature selection mechanism proposed by ASIKAINEN and MANNISTÖ along with Nivel is realized in a way that outputs all stable models where each model represents one feature configuration [AM09, p. 540]. The configuration task performed to find a valid configuration is actually a search problem that looks for actualized elements. In effect, *smodels* will output exactly the amount of stable models that correspond to the amount of configuration alternatives for a concrete feature model.
This approach to realize feature configuration has deliberately been put aside during implementation in CHAPTER 10. The process steps implemented in the DML environment start with selection and reduction of a given DML. This is necessary before transformation according to change operations, in order to grant changeOperations the property to be tailored by features. Since Nivel syntax is only created after transformation, the configuration cannot be part of *smodels*' execution.

Feature types: mandatory and optional. The feature model provided by the DML environment is simplistic. There are only mandatory and optional features. The next complexity in feature modeling contains alternative features, i.e. features related to each other where exactly one must be selected, the others deselected.
Since this thesis has no focus on product line engineering as a research discipline, the complexity has been reduced. This was decided to prevent complex handling of feature interactions, which were pointed out in CHAPTER 3.2.4.

Configurability is provided once. Another aspect in CHAPTER 3.2.4 is that configurability must be managed at all levels of abstraction. The DML concept does only allow configurability in the first step of the DML environment, so there is the restriction that configuration during project runtime is not possible, resp. unsupported.

Selection support. In complex development models, there may reside a large amount of features for full configuration flexibility. This flexibility comes with the implication of a more complex selection process. This process may be supported by definition of a surrogate model that can be used instead of direct access to the features. This surrogate model encapsulates the features into a smaller model that reduces the amount of decisions a DML manager has to take during configuration. The V-Model XT provides such a surrogate model for project managers. During tailoring, the project manager does not directly select process modules. Instead, he specifies project characteristics by answering to a set of questions [KN05]. Each question encapsulates a set of process modules. Thus, the project manager does

only need to answer some domain-specific questions and the surrogate model takes responsibility for the configuration of the V-Model XT.
GNATZ provided a similar, more generic approach for model configuration [Gna07, p. 170]. The approach relies on the selection of modules that have different kinds of relations to what is referred to in this thesis by features. These relation types allow the definition of aggregation, alternatives, and mutual exclusiveness of features.
Such a surrogate model for feature selection could enhance the configuration process of a DML with usability.

11.2.2. Extensibility

Extensibility is more an inherent property of a DML architecture than a mechanism. The creator of the DML architecture must be aware that the creation of association classes allows extensibility of a DML at a later point in time. Thus, during design of a DML architecture, it must be decided for each association whether this association and the related elements are intended to be extensible.
For 1-to-1 relations, it is unlikely that such an association should be realized by an association class just for the sake of extensibility. The reason is that in general, a 1-to-1 relation cannot be rearranged with additional participants. However, it may be reasonable to make use of an association class in such a case. But then, we would do so in order to gain access to additional benefits that arise with the usage of an association class like the possibility to place additional information related to an association. We would not do it to make the model more extensible.
Generally, a core asset engineer has to decide for each association, if the association is an integral part of one of the participants, or if it represents an independent relationship that may be created without direct involvement of the related elements. If the second case is given, and extensibility is a factor, then the relationship should be realized using an association class.

11.2.3. Modifiability

Modifiability has the most potential to actually create variants that largely differ from the original model. Configurability and extensibility are rather restricted mechanisms that do only allow slight deviations in the contents and its structure. Modifiability on the other hand is able to provide a toolkit of operations that allows the alteration of almost any content within a model. Because of this property, modifiability was the reason to add a restriction mechanism to the DML concept.

Transformational language needed. Modifiability is not only the most potent mechanism in this thesis, but also the only variability mechanism that cannot be realized within the usage of a logic programming language like Nivel. The reason is that it implies model to model transformations, which cannot be formulated with this language.

Thus an additional method for model transformations is needed. Nivel only provides an analytical approach for model evaluation.

Upgrades of the reference model. Since modifiability allows the organization engineer to exchange a reference model with a newer version, the DML concept bears some potential. At the least, there is a potential benefit because a DML variant is less dependent from a particular version of its reference DML. Small changes, bug fixes and additions are likely to not create much difficulties during exchange of a reference model.
A possible downside may be structural changes and massive changes in regard to contents, as well as meta-model changes. Such changes demand a keen eye for the needed adaptations in the extension model and likely some effort to create a consistent DML variant of an upgraded version of the reference model.
For the DML concept, there has been no major case study investigating the effect of an upgrade of a reference model to a customized variant. For restricted examples, the mechanism suited well, as there were only few conflicts to attend to.

Scaling change operations. The granularity of change operations may be very different. A change operation may only affect atomic entities like attribute values and reference, but the concept may as well be used to transform large portions of a model, like model refactoring, for example.
There are no recommendations provided in this thesis on how to design change operations and on how to evaluate their effect.
The granularity of the four variability operations provided in SPEM is quite coarse [OMG08b]. This results in the disadvantage, that "if there are parts of the generic process that you want to use in the description of the project specific task, there needs to be manual duplication of the content" [ZTSJ09].
During design of change operations, it must be considered whether existent properties are simply overwritten, or embedded into the varied state.

Pre- and postconditioned change operations. The constraint restriction mechanism proposed in this thesis places its focus on invariants regarding the interpretation of the varied state after configuration, extension, and modification. Pre- and postconditions for change operations were excluded, but may be an interesting aspect to be factored in.
Conditions might add more control over the execution process and enable a more directed application of changes.

Increasing model complexity. Usage of change operations adds elements to a model. Such elements have unique identifiers and need to be managed along with the model elements representing a development model's actual content.

Hence, the amount of elements and the management overhead is increased with the proposed modifiability concept.
It is unclear, and has not been investigated in this thesis, if adopting the modifiability approach is applicable in the long run, i.e. if it scales with the amount of variants and versions of a root reference model.

Consistent use of change operations. Until now, it was implicitly assumed that change operations represent a means of the organization engineer to modify a model. This assumption unintentionally disqualifies the core asset engineer as a user of change operations. This restriction is not necessary, though.
A core asset engineer may use change operations for the further development of a reference model, instead of applying direct modifications. Yet, the implications of such an approach have not been investigated.
In addition, using change operations for model evolution further contributes to model complexity as it is described above.

11.3. Review on variant restriction

As described in CHAPTER 4, two ways to realize variant restriction were identified: transformational and analytical restriction.

11.3.1. Transformational restriction

The usage of change operation has the inherent property to provide transformational restriction.
An organization engineer may only use change operation types that were provided within the DML architecture of the DML that is to be adapted. Along with a write-protection of the reference model, transformational restriction guarantees that the variant space is restricted to a specific class of variants.
This does not mean that the variant space is limited, but it may be. A limited variant space can be achieved in case there are only change operation types in a DML architecture which operate on discrete values, like enumeration types.
As observed in CHAPTER 4.1, a change operation may imply either a valid result model in respect to the requirements of the core asset engineer, or an invalid model. When model variants may be created that are deemed invalid, then we need an additional mechanism to check if a created variant is actually invalid, or not. This mechanism is realized by analytical restriction.

11.3.2. Analytical restriction

As we have seen in CHAPTER 4.2, analytical restriction covers syntax restriction, constraint restriction, and manual restriction. Correct syntaxes are regarded as a

prerequisite and are not covered in this thesis. Manual restriction is informal and bound to domain-specific requirements. Since actual DML realizations were largely out of the focus here, manual restriction was not covered, likewise.
Constraint restriction has been realized as a concept relying on the model element c̄onstraint in the DML framework.
A c̄onstraint can either be satisfied by a model, or not. There is a specialization of c̄onstraint, namely t̄ypeConstraint, that can be used to formulate constraints over types.

Inviolability of the DML architecture. The logic language approach using predicates to identify whether model restrictions are satisfied bears a risk for manipulation. If in an adaptation, it was made possible to add a mere fact stating that a c̄onstraint was fulfilled that is actually not fulfilled, then the resolving of the constraints may create a wrong answer set.
Thus, it must be assured that an organization engineer is not able to let additional manipulative facts into the model.
For instance, when using Nivel as modeling language for DML framework, DML architecture, and DML, there is a way to circumvent the mechanism for constraint restriction, if care is not taken. In order to feign that a c̄onstraint *theConstraint* is satisfied, the following predicate must be true:

```
constraintIsFulfilled(theConstraint).
```

In an environment that has not been manipulated, this predicate can only be evaluated to true, if *theElement* actually is fulfilled. Nevertheless, it may be that in the DML, a fact is added manually that sets exactly this predicate to true for *theConstraint*.
To prevent this, usage of the predicate *constraintIsFulfilled* must be disallowed in the DML. If this condition can be assured, then there is no risk that a constraint is satisfied if it actually is not.

Definition of constraints. The constraint mechanism relies on the specialization of c̄onstraint and on associated semantics provided by a constraintIsFulfilled rule in Nivel syntax in the DML architecture.
To define such semantics is a task performed by the core asset engineer. The core asset engineer must be able to specify rules in Nivel syntax. This must be done manually and a distinct domain knowledge is needed.
With the transformations described in CHAPTER 10, the organization engineer does not need to be able to create Nivel syntax. But he must be able to interpret the output of *smodels*. The result is a set of stable models. If the set is empty, the input model after configuration and transformation is invalid. Otherwise, each stable model in the result set represents a solution to the tree of restrictions spanned by the restriction terms.

11.4. Comparison with stated problems

In this thesis, we saw a concrete concept for realizing a variability framework, namely a DML framework. With this concept, we can now have a look into the solution of the problems stated in CHAPTER 2.

Handling Problem 1: Updates of the reference model

Exchanging a reference model with a newer version is made possible by the DML approach. An organization engineer may upgrade his customized variant of the reference model by file exchanges.
However, in some cases, problems may arise when exchanging model contents that are referenced within the organization-specific extension model. In the example of SECTION 7.7, no conflicts interfered while upgrading the reference DML to a newer version. But in general, conflicts may arise during upgrade of a standard DML that is referenced by an extension model. Possible conflicts are:

1. Deleting a model element from the reference model results in a conflict if the extension model or any of its extensions have outgoing references to it. The organization engineer must identify those conflicts and repair them manually. In some cases, the organization-specific DML extension or parts of it have to be redesigned.
2. If any element is changed in the reference model, while in the customized model a model element exists that refers to it, it must be checked if the implied semantics is still within the original meaning.
3. Semantic duplicates may arise during evolution of either model branch. I.e., a customized model might define new elements and afterwards, a newly released reference model version might contain the same additions with slight deviations. In this case, an automatic upgrade of the customized model will result in duplicate elements with similar or the same meaning.

The gravity of these conflicts is increasing from the first to the last. The first conflict threatens model integrity, but can easily be located. A manual fix of the dead-ended reference is necessary in all cases. This conflict type is deemed less severe, since if this conflict arises, we know there is an error that has to be fixed in all cases.
The second conflict is similar, but more difficult to locate. We can automatically generate a list of affected model elements, but there is no help in deciding whether an identified location actually contains a conflict. So here we do not only need a manual solution if a conflict arises, but also a manual evaluation whether a conflict is present.
The last conflict is the most severe because there is no trivial way to automatically locate duplicates within different model branches. So in this case, there is no list to be taken and evaluated manually, but the whole change list.

In conflicts 2 and 3, the rationale of the addition of development model contents may be used to better confine the differences between the alternative additions and thus manually derive a solution for the conflicts.

In short, if the first conflict arises, this always indicates an error. The second conflict may be erroneous, but it may also appear in correct models. It has to be checked manually. The third conflict however can not automatically be identified. In general, none of these conflict can be resolved automatically without human intervention.

Handling Problem 2: Conformity of an organization-specific development model

In the DML concept, two approaches were integrated that regard conformity of a DML variant to domain-specific restrictions: transformational and constraint restriction. The other analytical restriction types were not considered in this thesis.

Transformational restriction. For the realization of transformational restriction, the DML framework provides a class ChangeOperation which is to be specialized in a concrete DML architecture. This architecture is a development subject of the core asset engineer, the original model owner. Thus, the owner has the possibility to restrict the change operations available for a customized DML. This effectively restricts the quality of changes that can be applied to the model, provided that the same meta-model of change operations is used and the reference model remains unchanged.

In order to have the possibility to automatically allow only those changes that lead to a conform model, the core asset engineer must be aware that the tool set of change operations provided must be kept as small as possible, but as large as required.

Constraint restriction. It is difficult to determine an optimal size of the change operation toolkit in a DML architecture. Constraint restriction has been provided to add a more explicit mechanism to control the creation of model variants. This was done by explicit addition of constraints over a model, placed within the model itself.

The design of the DML framework allows the addition of constraints directly into the model. This provides the possibility to both the core asset engineer and the organization engineer to formulate restrictions over a model.

Such restrictions must be implemented in a way that they are able to filter out models that do not conform to domain-specific requirements.

Handling Problem 3: Localization of changes

Traditional approaches to track the differences between two versions of a model either imply the manual creation of a change list during development, or the *a posteriori* analysis of both versions [Arm08, p. 8].
After having modeled explicit change operations, these operations can easily be used to infer a list of differences between the original and the varied result. An instance of a change operation provides the parameters that have to be used for its execution, as well as the semantics defined by its concrete type, which have to be interpreted in a predefined way. The combination of parameters and semantics of (instances of) change operations can be regarded as the difference between two versions of a model.
Since a change list is nothing else than a description of differences, and change operations are a descriptive way to express changes, change operations can be translated into a human readable change list.

Handling Problem 4: Loss of the rationale of changes

Change operations provide a possibility to explicitly describe modifications. In addition, if we document the rationale of the change along with the operation, we can assure that the information about *why* a certain change has been done is not lost.
This way, the following information can be attached to each change for later reference: motivation of the change, pro&contra of the change, and discussion of pro&contra.
A meta-model for the rationale of changes and a proposition on how to relate this to a model was proposed by OCAMPO and SOTO [OS07].

11.5. Implementation technologies

Three technologies were used for the implementation of the DML concept: UML, XSLT, and Nivel. UML and Nivel were used as modeling languages, while XSLT has been used both to translate UML into Nivel syntax, and to operationalize change operations.

11.5.1. Modeling language

The DML framework has been described using UML class diagrams. UML is a very common modeling language and its class diagrams are suited well to describe static model structures. The popularity of UML has been one reason to use UML for modeling DML's.
Another reason is the XMI specification offered by the OMG [OMG07a], which explicitly supports UML. The specification allows to make use of a standardized

file format for persistent storage of UML models, which was needed to be able to further process over DML models.
Since the exemplary DML architecture and DML rely on the meta-model provided by the DML framework, is it obvious to use UML to describe these, too.
In the DML environment's course of execution, the UML models representing a DML are transformed into Nivel syntax, which is used as another language for DML model representation. The main reason for usage of Nivel lies in its capability to be used to reason over a model's properties (see constraint checking below).
In addition, it is an interesting case of a model relying on the OMG MOF layers [OMG06] that is transformed into a model making use of deep instantiation [AK01] and strict meta-modeling [AK02]. This is enabled by transformation of a specialization layer into an instantiation layer. This layer is represented by the DML architecture.
Using specialization leaves the crossing between DML framework and DML architecture diffuse from the modeling view. Instantiation on the other hand keeps the layers strictly separated.

Alternatives. In respect to a largely definite semantics and a file format that can be interpreted, edited and interchanged by common modeling tools, an alternative to UML is usage of the Eclipse Modeling Framework (EMF) [Ecl10b].
The tool support for EMF orientates largely on the application scenario of software and systems development, and there exist less convenient model editors, so UML was chosen for model representation.
Neither UML, nor EMF support an explicit realization of deep instantiation and strict meta-modeling.

11.5.2. Model transformations

There are some alternatives to Extensible Stylesheet Language Transformations (XSLT) for the realization of model transformations. But let's first take a look at the different application scenarios for model transformations in the DML concept:

1. Reduction of a UML model according to selection information.
2. Transformation of a UML model according to change operations.
3. Translation of a UML model to a rule checking language.

The first two use cases represent UML to UML transformations, while the third case implies translation of UML to another language. In order to keep the amount of technologies used for realization of a DML environment small, it appears reasonable to use a single transformation language for all three steps. XSLT is a technology provided by W3C that can do this [W3C99].

Alternatives. QVT is an OMG specification covering model to model transformations [OMG08a]. This specification has a prominent implementation named Atlas Transformation Language (ATL) [Ecl10a]. ATL is able to transform both MOF and EMF based models. But since ATL is restricted to model to model transformations, in regard to application scenario 3, ATL has not been chosen as transformation technology.
Other model transformation technologies like graph based transformations [TC06] need a graph representation of the model, which would include at least one further transformation into the process.
In the V-Model XT, the operationalization of change operations is implemented in Java. The transformation tool, namely the V-Model XT Editor, is able to merge reference and extension models and to interpret change operations residing in the resulting model. Change operations are not implemented atomically. They are built from a set of basic operations like exchange of element references or concatenation of string attributes.
Another existing operation-based approach is EMFStore [KH10a]. Although its focus lies on recording of change in contrast to change modeling, the underlying concepts are well suited to be used as a modifiability framework for EMF models.

11.5.3. Constraint checking

Nivel has been used to verify both the model integrity of the configured and modified DML models during development of this thesis, as well as a means to check constraint restriction. It is a language with a native support to express UML-like models.
Model integrity is verified by transformation of the UML models to Nivel syntax for DML framework, DML architecture, and DML. Nivel will only find stable models if the modeling adhered to the considerations described in CHAPTER 10.1. This assures a proper UML modeling of the DML models.
As a logic language, Nivel allows the definition of arbitrary constraint checks by provision of constraintIsFulfilled predicates in the DML architecture.
These constraint checks are related to actual instances of the model in respect to both elements that have to fulfill certain criteria, as well as elements that are used to decide whether such criteria are fulfilled. This provides a flexible mechanism to place restrictions within a model.
Once a meta-model of constraints is created in the DML architecture, the core asset engineer, as well as the organization engineer, can freely arrange terms consisting of constraints to model variant restriction.

Alternatives. OCL [OMG10a] provides the possibility to formulate constraints over a model, too. But OCL constraints are a static part of the meta-model, in contrast to constraints being placed within the model. I.e., without change of the

DML architecture, the addition of restrictions to the model would not be possible with OCL.
Other logic languages besides Nivel were neglected because of Nivel's native capability to reason over UML-like models.

12. Conclusion

The exemplary development model line (DML) described in CHAPTERS 7 and 9 is an instance of a configurable, extensible, and modifiable development model. The definitions of *configurability*, *extensibility*, and *modifiability* discussed in CHAPTER 3 are extracted from existing frameworks and technologies that claim to have a positive influence on the variability of their subjects.
The implementation of the DML concept explained in CHAPTER 10 is a continuous example for a realization. It covers...

... modeling a DML along with its architecture and the DML framework in UML,

... the possibility to extend and configure a DML model,

... operationalization of modifiability of DML models, and

... verification that a resulting development model is within a predefined admissible variant space.

With the DML concept, this thesis provides support for the customization and maintenance of development models. The originator of a DML, i.e. the core asset engineer, is able to scale the variability of 'his' DML. He does this by arranging configurability, extensibility, and modifiability. In addition, it is possible to define constraint restrictions over a model. These restrictions confine the amount of variants that can be created using variability.
A summary of the main contributions of this thesis is shown in TABLE 12.1. For each contribution, the related goal(s) and a reference to the chapter(s) covering the matter is provided. Contributions 2–5 are related to the goals stated in CHAPTER 1.1. Contribution 1 is needed for spanning the problem domain and contribution 6 is used for an evaluation of soundness and consistency of contributions 2–5. In addition it discusses semantical issues of modifiability and variant restriction.
As shown in CHAPTER 11, there are several aspects that need attention when the DML concept is adopted to a development model. These are subject to future work.

- Configurability is restricted to mandatory and optional features. More elaborate feature types were neglected, as well as questions concerning overlapping features, resp. the effect of features with overlapping contents.
- Change operations have to be added to a DML architecture. The question is: which operations are feasible? The core asset engineer has to define the exact toolkit of change operations, but has no support in how to do it right.

	Contribution	Goal(s)	Chapter(s)
1.	Proposition of a comprehensive definition for the term 'development model'.	–	2
2.	Clarification of three kinds of variability, namely configurability, extensibility, and modifiability.	1, 2	3, 6
3.	Restriction of the variant space of development models with domain-specific rules and model-specific constraint criteria.	3	4, 8
4.	Definition of an abstract framework for the creation of variable development models and provision of a conceptual realization example making use of this framework.	4	7
5.	Integration of this framework with the introduced concepts for variant space restriction.	5	9
6.	Implementation of the theoretical concept and its semantics with UML, XSLT, and the logic programming language Nivel.	–	10

Table 12.1: Main contributions of this thesis

- It has to be shown how powerful the constructs actually are we can create with the constraint restriction framework. Boolean terms of restrictions can be created but it has not been investigated which classes of restrictions are possible, which feasible, and which impossible.
- The concept includes extensibility, configurability, and modifiability as these terms were defined in CHAPTER 3. Other aspects of variability were not included.
- Model complexity increases with explicit usage of change operations and constraint restriction. The influence on scalability of development models and their versions and variants has to be investigated.
- It has yet to be shown whether the applicability of change operations scales without a more explicit concept for change operation ordering.
- A development model created with the DML concept may be regarded as a meta-model for a particular project. There are no implications made in this thesis on how the enactment of a development model may be realized.
- The presented implementation of the concept uses two very different technologies for modifiability and constraint restriction. An integrated language able to both transform a given model, as well as evaluate the proposed constraint framework may increase the applicability of the concept.

The proposed DML concept is a promising approach to realize variability of development models. It neatly combines variability mechanisms observed in various technologies, development models, and frameworks into one concept. With these mechanisms, it offers detailed support to both the core asset engineer and the organization engineer in respect to flexibility of adaptations and to conformity of

the created results.

Appendix

Model transformations for the DML environment

1. DML reduction: XSL implementation

In this section, the XSL script used for the reduction of a DML is described. Note that if there is a line numbering gap, this is in all cases due to an empty line in the original reduce.xsl file.
The first two lines contain the usual XSL stylesheet header. Since we are operating on XMI models, the XMI namespace is included.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform" xmlns:xmi='http://schema.omg.org/spec/XMI/2.1'>
```

The result of the reduction is an UML XMI file, so the output format is set to XML. The file encoding of the UML models is set by MagicDraw to UTF-8, so the output is created using the same encoding.

```
4     <xsl:output method="xml" encoding="UTF-8"/>
```

In the next lines, the script initializes a set of variables used later on to identify elements to be removed from the UML model. First, all instances of belongs_to are identified, then stored in the XSL variable *belongsToInstances*.

```
6     <xsl:variable name="belongsToID" select="//packagedElement[@name='
        belongs_to']/@xmi:id"/>
7     <xsl:variable name="belongsToInstances" select="//packagedElement[
        classifier/@xmi:idref=$belongsToID or @classifier=$belongsToID]"/>
```

Now, all instances of feature are identified and stored in the XSL variable *featureInstances*.

```
8     <xsl:variable name="featureFrameworkElementID" select="//
        packagedElement[@name='feature']/@xmi:id"/>
9     <xsl:variable name="featureInstances" select="//packagedElement[
10              classifier/@xmi:idref=$featureFrameworkElementID or
                  @classifier=$featureFrameworkElementID
11          ]"/>
```

Finally, all instances of featureDeselector are identified and gathered using the variable *featureDeselectorInstances*.

```
12    <xsl:variable name="featureDeselectorID" select="//packagedElement[
        @name='featureDeselector']/@xmi:id"/>
```

```
13      <xsl:variable name="featureDeselectorInstances" select="//
            packagedElement[
14                  classifier/@xmi:idref=$featureDeselectorID or @classifier
                        =$featureDeselectorID
15              ]"/>
```

Having these variables at hand, the next lines store in the variable *deselectedFeatures* all feature instances that were target of any deselects association instance.

```
16      <xsl:variable name="deselectedFeatures" select="$featureInstances[
17          @xmi:id=//packagedElement[
18              slot/value/@instance=$featureDeselectorInstances/@xmi:id
19          ]/slot/value[
20              @instance!=$featureDeselectorInstances/@xmi:id
21          ]/@instance
22      ]"/>
```

The next variable, namely *selectedFeatures*, is determined straightforwardly by taking the set of *featureInstances* and subtracting all features that were deselected.

```
23      <xsl:variable name="selectedFeatures">
24          <xsl:choose>
25              <xsl:when test="not($deselectedFeatures)">
26                  <xsl:for-each select="$featureInstances">
27                      <ids>
28                          <xsl:attribute name="xmi:id" select="@xmi:id"/>
29                      </ids>
30                  </xsl:for-each>
31              </xsl:when>
32              <xsl:otherwise>
33                  <xsl:for-each select="$featureInstances[@xmi:id!=$
                        deselectedFeatures/@xmi:id]">
34                      <ids>
35                          <xsl:attribute name="xmi:id" select="@xmi:id"/>
36                      </ids>
37                  </xsl:for-each>
38              </xsl:otherwise>
39          </xsl:choose>
40      </xsl:variable>
```

The next variable *deselectedElements* contains all those model elements that do not have any belongs_to association to any selected feature.

```
41      <xsl:variable name="deselectedElements" select="//packagedElement[
42          @xmi:id=$belongsToInstances[
43              not(slot/value/@instance=exslt:node-set($selectedFeatures)/ids
                    /@xmi:id)
44          ]/slot/value/@instance
45      ]"/>
```

The last variable needed throughout the rest of the script is *deselectedElementRelations*. It parses through all association instances and looks for any relations to any of the *deselectedElements*. Such an association is marked as being deselected, too.

```
46      <xsl:variable name="deselectedElementRelations" select="//
            packagedElement[
47          slot/value/@instance=$deselectedElements/@xmi:id
48      ]"/>
```

Now, the script is actually starting. The obligatory '/' root template is simply calling to apply all other templates to the root node.

```
50      <xsl:template match="/">
51          <xsl:apply-templates/>
52      </xsl:template>
```

Here is the script with the lowest priority. If a node matches no other template, then this template is executed. The script does nothing else than copying the actual node into the result document with all its attributes, then check all direct sub-nodes against all templates in the script. In the usual case, this script is called recursively until there either are no more sub-nodes, or one of the other templates with a higher priority apply.

```
54      <!-- go through the actual node and copy it and all its attributes,
55           unless the node is addressed by another template!-->
56      <xsl:template match="node()" priority="1">
57          <xsl:copy>
58
59              <!-- copy all attributes of the actual node -->
60              <xsl:copy-of select="@*"/>
61
62              <!-- call the templates recursively -->
63              <xsl:apply-templates/>
64          </xsl:copy>
65      </xsl:template>
```

If one of the templates below apply to a node, then the copy-template described above is not executed. All of the following templates generate no output, effectively removing a node from the UML model. The first template matches if the ID of the node tested against the templates is in the list *featureDeselectorInstances*. In effect, this removes all instances of fFeatureDeselector from the model. They are no longer needed for the further process in the DML environment.

```
67      <!-- remove all nodes that have a featureDeselector as classifier -->
68      <xsl:template match="node()[@xmi:id=$featureDeselectorInstances/
            @xmi:id]" priority="2"/>
```

The next template removes an element from the model, if its ID is stored within *deselectedFeatures*. This removes all fFeature instances from the model that were deselected.

```
70      <!-- remove all feature nodes that have been deselected -->
71      <xsl:template match="node()[@xmi:id=$deselectedFeatures/@xmi:id]"
            priority="3"/>
```

If a node's ID is covered by *deselectedElements*, then it is not written to the output model. In effect, this removes all instances with no fBelongs_To relationship to any selected fFeature.

```
73      <!-- remove all modelElement instances that do not have a 'belongs_to'
            relation to any feature that resides in the model -->
74      <xsl:template match="node()[@xmi:id=$deselectedElements/@xmi:id]"
            priority="4"/>
```

Finally, in order to reestablish model consistency, all internal association references to elements that were removed from the model are removed as well.

```
76      <!-- remove all relations to deselected modelElement instances  -->
77      <xsl:template match="node()[@xmi:id=$deselectedElementRelations/
            @xmi:id]" priority="5"/>
78  </xsl:stylesheet>
```

2. DML transformation: XSL implementation

The transformation of DML models, i.e. the execution of all change operations in the model, is following a generative approach. For each changeOperation instance identified in the DML model, a separate XSL script is generated and executed. In general, each generated XSL script takes a DML model as input and outputs the same model, with changes made according to the exact type of the change operation. The resulting DML model is then looped through the next change operation script until all change operations were executed.
For the generation of the change operation effect scripts, an XSL script transform.xsl was written. It is described in this section. It describes the generation of XSL scripts that create the effect of a change operation of type *changeProductResponsibility*, as it was introduced as example in CHAPTER 7.4.
After the description of this script, an exemplary generated effect script is explained. The script is implementing the change of responsibility for the product qaManual. This particular change operation was described as an example in the extension model shown in CHAPTER 7.4.

transform.xsl This script is implementing the generation of effect scripts for change operations. Note that only the semantics for changeProductResponsibility is provided. If another specialization of changeOperation is added to a DML architecture, the semantics of the operation has to be implemented in this generator script.
The first two lines contain the usual XSL stylesheet header. Since we are operating on XMI models, the XMI namespace is included.

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
        Transform" xmlns:xmi='http://schema.omg.org/spec/XMI/2.1'>
```

The result of the transformation is an XSL file, so the output format is set to XML. The output encoding is set to ISO-8859-1, since this is the encoding chosen for the

XSL scripts at hand. For a better readability of the result file, the XSLT interpreter is told to indent the resulting XML code.

```
4     <xsl:output method="xml" encoding="ISO-8859-1" indent="yes"/>
```

For better access to certain model elements, some variables are initialized. First, the framework element changeOperation is identified and stored in the variable *changeOperationFrameworkElement*. This element is used to identify all specializations of changeOperation in the architecture. These are stored as *changeOperationArchitectureElements*. Finally, with the knowledge of all change operation types, *changeOperationInstances* is set to all instances of these types.

```
6     <xsl:variable name="changeOperationFrameworkElement" select="//
          packagedElement[@name='changeOperation']"/>
7     <xsl:variable name="changeOperationArchitectureElements" select="//
          packagedElement[
8                 generalization/@general=$changeOperationFrameworkElement/
                      @xmi:id
9                 or
10                @generalization=$changeOperationFrameworkElement/@xmi:id
11          ]"/>
12    <xsl:variable name="changeOperationInstances" select="//
          packagedElement[
13                classifier/@xmi:idref=$changeOperationArchitectureElements
                      /@xmi:id
14                or
15                @classifier=$changeOperationArchitectureElements/@xmi:id
16          ]"/>
```

Now the root template begins. Output generation starts here with an XML element named 'xsl:stylesheet'. The element is provided with the XMI namespace and an attribute 'version' denoting the XSL version 2.0. When executed, we can see that this XSL code generates the XSL stylesheet header for an effect script. For the further understanding of the documented code, it may be useful to take looks at the exemplary effect script shown below the description of the generator script (see 163).

```
18    <xsl:template match="/">
19
20        <xsl:element name="xsl:stylesheet">
21            <xsl:namespace name="xmi">http://schema.omg.org/spec/XMI/2.1</
                  xsl:namespace>
22            <xsl:attribute name="version">2.0</xsl:attribute>
```

The following lines statically create an XML node <**xsl:output** method="xml"indent="no"/>. We want that the resulting script is able to transform UML XMI models, so we tell it to work on XML data and to not interfere with the indention it runs across.

```
23            <xsl:element name="xsl:output">
24                <xsl:attribute name="method">xml</xsl:attribute>
25                <xsl:attribute name="indent">no</xsl:attribute>
26            </xsl:element>
```

Like all XSL stylesheets, the generated script must contain a root template matching the root node '/'. All our root template shall do is check for all child nodes of the root if any of the other templates apply.

```
28          <!-- generate the root template -->
29          <xsl:element name="xsl:template">
30              <xsl:attribute name="match">/</xsl:attribute>
31              <xsl:element name="xsl:apply-templates"/>
32          </xsl:element>
```

The next lines create code that is identical to the copy-code we saw in lines 56–65 of script reduce.xsl. The result is that like above, the template with the lowest priority simply copies a node and its attributes, then applies all templates to its children.

```
34          <!-- generate the generic template that is copying a node and
                all its attributes and subnodes, unless there exists a
                template with a higher priority for this node -->
35          <xsl:element name="xsl:template">
36              <xsl:attribute name="match">node()</xsl:attribute>
37              <xsl:attribute name="priority">1</xsl:attribute>
38              <xsl:element name="xsl:copy">
39                  <xsl:element name="xsl:copy-of">
40                      <xsl:attribute name="select">@*</xsl:attribute>
41                  </xsl:element>
42              <xsl:element name="xsl:apply-templates"/>
43              </xsl:element>
44          </xsl:element>
```

Since for every change operation, a separate effect script is generated, we need to identify the operation to be executed with the script actually created. The identified instance of c̄hangeOperation is stored in the variable *theOperation*.
For simplicity, as it was specified in CHAPTER 7.6, this particular change operation is always the first change operation that can be found in the model. Since an effect script is always not only executing the change indicated by a change operation, but also removing the change operation instance from the model in the same step, a call to the first change operation in the model will always result in another change operation on repeated executions.

```
46          <!-- there is exactly one operation to be performed in the
                next step -->
47          <xsl:variable name="theOperation" select="$
                changeOperationInstances[1]"/>
```

Up until now, the script contained generic XSL code used to create the general lines that are common to all effect scripts. The rest depends on the particular change operation. The following script code depends on the concrete type of the instances identified by *theOperation*. For every specialization of c̄hangeOperation placed in the DML architecture, a concrete semantics must be provided within this XSL script. What code is to generated is decided by the type of the change operation.

```
49          <!-- generate the xsl code for the very first change operation
                , identified clearly by "$theOperation" -->
```

```
50              <xsl:choose>
```

We check now for the type of *theOperation*. If it is an instance of the change operation changeProductResponsibility, then generate the corresponding XSL code. As described in CHAPTER 7.4, such an instance is intended to replace a role reference in a responsibility instance.

```
51                  <!-- when the change operation is an instance of
                        changeProductResponsibility -->
52                  <xsl:when test="'changeProductResponsibility' = $
                        changeOperationArchitectureElements[@xmi:id=$
                        theOperation/classifier/@xmi:idref or @xmi:id=$
                        theOperation/@classifier]/@name">
```

For better readability of the effect script, a comment is generated informing about the type of the identified change operation and its ID. The <**xsl:text**> node is merely used to rectify output indention, since the indention created by the XSLT interpreter Saxon is damaged after insertion of a comment when using <**xsl:comment**>.

```
53                      <xsl:comment> Templates for a change operation of type
                            'changeProductResponsibility'. The instance has
                            the id '<xsl:value-of select="$theOperation/@xmi:id
                            "/>' </xsl:comment><xsl:text>
54      </xsl:text>
```

First, the DML architecture is parsed for the elements representing the responsibility associations. In particular, we need its association end source, as this is the indicator pointing to a role. This end is stored in the variable *roleEnd*.

```
55                      <!-- attributes of the responsibility architecture
                            element -->
56                      <xsl:variable name="responsibilityArchitectureElement"
                            select="//packagedElement[@name='responsibility']"
                            />
57                      <xsl:variable name="roleEnd" select="$
                            responsibilityArchitectureElement/ownedEnd[@name='
                            source']"/>
```

The following lines are used to determine the values of two variables. They cover *relevant_modifiesResponsibilityInstance* and *relevant_targetInstance*. The first variable, *relevant_modifiesResponsibilityInstance*, is the very instance of type modifiesResponsibility that is related to the change operation *theOperation*. *relevant_targetInstance* is the instance of responsibility that is attached to the association end target in that instance. So in effect, these lines determine the responsibility instance that is affected by the change operation. It can then be referenced by the variable *relevant_targetInstance*.

```
59                      <!-- find target instance -->
60                      <xsl:variable name="
                            modifiesResponsibilityArchitectureElement" select="
                            //packagedElement[@name='modifiesResponsibility']"/
                            >
```

```
61                      <xsl:variable name="modifiesResponsibilityInstances"
                            select="//packagedElement[
62                              classifier/@xmi:idref=$
                                    modifiesResponsibilityArchitectureElement/
                                    @xmi:id
63                              or @classifier=$
                                    modifiesResponsibilityArchitectureElement/
                                    @xmi:id
64                          ]"/>
65                      <xsl:variable name="operationArchitectureElement"
                            select="$modifiesResponsibilityArchitectureElement/
                            ownedEnd[@name='operation']"/>
66                      <xsl:variable name="targetArchitectureElement" select=
                            "$modifiesResponsibilityArchitectureElement/
                            ownedEnd[@name='target']"/>
67                      <xsl:variable name="
                            relevant_modifiesResponsibilityInstance" select="$
                            modifiesResponsibilityInstances[
68                              slot[
69                                  @definingFeature=$
                                        operationArchitectureElement/@xmi:id
70                              ]/value/@instance=$theOperation/@xmi:id
71                          ]"/>
72                      <xsl:variable name="relevant_targetInstance" select="
                            //packagedElement[
73                              @xmi:id=$
                                    relevant_modifiesResponsibilityInstance/
                                    slot[
74                                      @definingFeature=$
                                            targetArchitectureElement/@xmi:id
75                              ]/value/@instance
76                          ]"/>
```

Similar to the procedure above, the following code first determines the value of *relevant_newRoleInstance*, which is the instance of newRole that is associated to *theOperation*. This instance is used to determine the instance attached to its param association end, which in effect is a role instance. In particular, this role instance is the role to replace the original role associated to *relevant_targetInstance*. It is referenced by the variable *relevant_paramInstance*.

```
78                      <!-- find parameter instance -->
79                      <xsl:variable name="newRoleArchitectureElement" select
                            ="//packagedElement[@name='newRole']"/>
80                      <xsl:variable name="newRoleInstances" select="//
                            packagedElement[
81                              classifier/@xmi:idref=$
                                    newRoleArchitectureElement/@xmi:id
82                              or @classifier=$newRoleArchitectureElement/
                                    @xmi:id
83                          ]"/>
84                      <xsl:variable name="operationArchitectureElement"
                            select="$newRoleArchitectureElement/ownedEnd[@name
```

```
                ='operation']"/>
85              <xsl:variable name="paramArchitectureElement" select="
                    $newRoleArchitectureElement/ownedEnd[@name='param']
                    "/>
86              <xsl:variable name="relevant_newRoleInstance" select="
                    $newRoleInstances[
87                      slot[
88                          @definingFeature=$
                                operationArchitectureElement/@xmi:id
89                      ]/value/@instance=$theOperation/@xmi:id
90                  ]"/>
91              <xsl:variable name="relevant_paramInstance" select="//
                    packagedElement[
92                      @xmi:id=$relevant_newRoleInstance/slot[
93                          @definingFeature=$paramArchitectureElement
                                /@xmi:id
94                      ]/value/@instance
95                  ]"/>
```

Now, the generation of XSL templates for the execution of the change operation will be defined. The necessary information is the responsibility instance to be changed and the role instance to be used as new responsible role. The responsibility has been identified above by *relevant_targetInstance*, while the role instance is identified by *relevant_paramInstance*.

The code below creates an <**xsl:template**> node with a match clause that looks for an association end within the association identified by *relevant_targetInstance*. The association end is identified by *roleEnd*, which effectively is the source association end of the responsibility.

Note that the match clause is effectively a concatenation of several strings. Since the resulting string must contain quotes, and the concatenation function needs additional quotes, too, some quotes contained within the strings are masked by ".

```
97              <xsl:element name="xsl:template">
98                  <xsl:variable name="matchClause">
99                      <xsl:value-of select="concat('value[../
                            @definingFeature=&#34;',$roleEnd/@xmi:id
                            ,'&#34; and ../../@xmi:id=&#34;',$
                            relevant_targetInstance/@xmi:id,'&#34;]')"/
                            >
100                 </xsl:variable>
101                 <xsl:attribute name="match">
102                     <xsl:value-of select="$matchClause"/>
103                 </xsl:attribute>
104                 <xsl:attribute name="priority">2</xsl:attribute>
```

Now, the actual replacement is stated. The following code generates code that, when the template applies, copies the identified node, all its attributes, and all its sub-nodes, but replaces the value of the 'instance' attribute with the ID of *relevant_paramInstance*. As mentioned above, this is the role instance denoted as

newRole by the change operation instance.

```
106                         <xsl:element name="xsl:copy">
107                             <xsl:element name="xsl:copy-of">
108                                 <xsl:attribute name="select">@*</
                                        xsl:attribute>
109                             </xsl:element>
110                             <xsl:element name="xsl:attribute">
111                                 <xsl:attribute name="name">instance</
                                        xsl:attribute>
112                                 <xsl:value-of select="$
                                        relevant_paramInstance/@xmi:id"/>
113                             </xsl:element>
114                             <xsl:element name="xsl:apply-templates"/>
115                         </xsl:element>
116                     </xsl:element>
```

The following line is representing the end of the condition that *theOperation* is an instance of changeProductResponsibility.

```
117                 </xsl:when>
```

This is the end of the choose clause, which was used to determine the type of the change operation. If more specializations of changeOperation are added to the DML architecture, then their semantics must be defined before this line embedded in a <xsl:when> node, just like shown beforehand for changeProductResponsibility.

```
118             </xsl:choose>
```

The rest of the script is generating another template that is looking for the very change operation instance that script was generated for. The code is intended to remove the change operation instance identified by *theOperation* from the DML model.

```
120             <xsl:choose>
121                 <xsl:when test="$theOperation">
122                     <!-- generate the code that removes the very change
                            operation "$theOperation" that is resolved in the
                            generated script -->
123                     <xsl:element name="xsl:template">
124                         <xsl:attribute name="match">node()[@xmi:id='<
                                xsl:value-of select="$theOperation/@xmi:id"/>']
                                </xsl:attribute>
125                         <xsl:attribute name="priority">3</xsl:attribute>
126                     </xsl:element>
127                     <!-- generate the code that removes all elements that
                            are related to the removed change operation -->
128                     <xsl:element name="xsl:template">
129                         <xsl:attribute name="match">//packagedElement[slot
                                /value/@instance='<xsl:value-of select="$
                                theOperation/@xmi:id"/>']</xsl:attribute>
130                         <xsl:attribute name="priority">3</xsl:attribute>
131                     </xsl:element>
132                 </xsl:when>
```

```
133                 <xsl:otherwise>
134                     <xsl:comment> No change operation was found in the
                            model. </xsl:comment><xsl:text>
135 </xsl:text>
136                 </xsl:otherwise>
137             </xsl:choose>
138         </xsl:element>
139     </xsl:template>
140 </xsl:stylesheet>
```

An exemplary effect script An exemplary script generated by the script above may look like the following code.
The header's content is identical to that of the reduce.xsl script: encoding UTF-8 and the same namespaces, since it is intended to generate the right XMI syntax.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
3                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4                 version="2.0">
```

The next lines are used to copy the input UML model to the output model. The output is exactly like the input, with the exception that for the templates described below, another behavior is defined.

```
5     <xsl:output method="xml" indent="no"/>
6     <xsl:template match="/">
7         <xsl:apply-templates/>
8     </xsl:template>
9     <xsl:template match="node()" priority="1">
10        <xsl:copy>
11            <xsl:copy-of select="@*"/>
12            <xsl:apply-templates/>
13        </xsl:copy>
14    </xsl:template>
```

The exceptional behavior is defined by three templates. The first template identifies an association end of type source that is contained within the responsibility instance that was targeted by the change operation.
Note that, like it was described above, " is masking a single quote. This means this code has the same effect like '.

```
15    <!-- Templates for a change operation of type '
          changeProductResponsibility'. The instance has the id '
          _16_6_2_284e05ec_1271941702875_40894_1309' -->
16    <xsl:template match="value[../@definingFeature=&amp;#34;
          _16_6_2_284e05ec_1269713897568_176111_1013&amp;#34; and ../../@xmi:id
          =&amp;#34;_16_6_2_284e05ec_1269958123338_858107_773&amp;#34;]"
17                  priority="2">
18        <xsl:copy>
19            <xsl:copy-of select="@*"/>
20            <xsl:attribute name="instance">
                  _16_6_2_284e05ec_1269859455832_166492_902</xsl:attribute>
```

```
21            <xsl:apply-templates/>
22        </xsl:copy>
23    </xsl:template>
```

The second template looks for the change operation instance and removes it from the resulting model.

```
24    <xsl:template match="node()[@xmi:id='
          _16_6_2_284e05ec_1271941702875_40894_1309']"
25                  priority="3"/>
```

Finally, the last template removes all associations with a reference to the removed change operation.

```
26    <xsl:template match="//packagedElement[slot/value/@instance='
          _16_6_2_284e05ec_1271941702875_40894_1309']"
27                  priority="3"/>
28 </xsl:stylesheet>
```

UML to Nivel translations

3. Transform DML framework package to Nivel

The following script implements the translation of a DML framework provided as a UML model in XMI format to Nivel syntax.
The script starts with a common XSL script header. Note that the output format is 'text'.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
       Transform" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3
4      <xsl:output method="text" encoding="ISO-8859-1"/>
5
6      <xsl:template match="/">
7          <xsl:apply-templates/>
8      </xsl:template>
```

There are two templates controlling the packages to be translated. The first template states that if any UML package has a name other than 'framework' or 'helpers', then the package is ignored, i.e. not included in the output generation. A package is identified by being a <packagedElement> node with an attribute value xmi:type of 'uml:Package'.
'framework' is the package holding the classes defined for the DML framework, while 'helpers' contains the class FeatureDeselector with its association deselects that were added in CHAPTER 10.1 in order to enable the relation of configuration information to the DML model.
If a package is found with the name 'framework' or 'helpers', then for all contained <packagedElement> nodes, the templates described below are applied.

```
10     <xsl:template match="packagedElement[@xmi:type='uml:Package' and @name
           !='framework' and @name!='helpers']">
11         <!-- do nothing, since this is not architecture relevant -->
12     </xsl:template>
13
14     <xsl:template match="packagedElement[@xmi:type='uml:Package' and (
           @name='framework' or @name='helpers')]">
15         %% create explicit framework elements %%
16         <xsl:apply-templates select="packagedElement"/>
17     </xsl:template>
```

The rule for the translation of classes is: look for all <packagedElement> nodes where the attribute xmi:type is set to the value 'uml:Class'. If this class has a parent class,

i.e. if generalization/@general is set to the ID of the parent, then output:

subclassOf_D(<name of the **class**>,<name of the parent **class**>).

Otherwise output:

topLevel_D(<name of the **class**>).
hasPotency_D(<name of the **class**>,3).

Note that the potency is always set to 3. This is due to the design decision[1] to place all the framework element on level 3.

```
19  <xsl:template match="packagedElement[@xmi:type='uml:Class']">
20      <!-- look for class definitions: classes at top level (->
            topLevel_D and hasPotency_D), and subclasses (->subclassOf_D)
            -->
21      <xsl:variable name="generalID" select="generalization/@general"/>
22      <xsl:variable name="generalName" select="//packagedElement[@xmi:id
            =$generalID]/@name"/>
23      <xsl:variable name="classID" select="@xmi:id"/>
24      <xsl:variable name="className" select="@name"/>
25      <xsl:choose>
26          <xsl:when test="$generalID">
27              subclassOf_D(<xsl:value-of select="@name"/>,<xsl:value-of
                    select="//packagedElement[@xmi:id=$generalID]/@name"/>)
                    .
28          </xsl:when>
29          <xsl:otherwise>
30              topLevel_D(<xsl:value-of select="@name"/>).
31              hasPotency_D(<xsl:value-of select="@name"/>,3).
32          </xsl:otherwise>
33      </xsl:choose>
```

Note that at this point of the code, we haven't yet left the translation of a class. I.e., the template is not yet closed. We have to translate the class's attributes now. This is done by a named call to the template 'translateAttribute', which will be described below.

```
35      <!-- look for attributes -->
36      <xsl:call-template name="translateAttribute"/>
37  </xsl:template>
```

The next template describes the translation of enumerations. If a <packagedElement> has an xmi:type value of 'uml:Enumeration', then the output for each literal (ownedLiteral) defined for this enumeration is:

contains(<name of the enumeration>,<value of the literal>).

Note that the enumeration itself has no representative in the Nivel syntax. This is implicitly done by definition of a value domain, which is created by using the name of the enumeration to span the domain of the enumeration values.

```
39  <xsl:template match="packagedElement[@xmi:type='uml:Enumeration']">
40      <xsl:variable name="classID" select="@xmi:id"/>
```

[1] See CHAPTER 10.4.4.

```
41          <xsl:variable name="className" select="@name"/>
42          <xsl:for-each select="ownedLiteral">
43              contains(<xsl:value-of select="$className"/>,<xsl:value-of
                    select="@name"/>).
44          </xsl:for-each>
45      </xsl:template>
```

Similar to the template translating simple classes, the translation of associations and association classes is covered by a separate template. The following template applies to <packagedElement> nodes where the value of the attribute xmi:type is either 'uml:Association', or 'uml:AssociationClass'.
In all cases, the template creates an association shortcut[2] according to the following pattern:

scn_association_D(<name of the **association**/associationClass>,3,<name of first role>,<**class** name of first role>,<name of second role>,<**class** name of second role>).

```
47      <xsl:template match="packagedElement[@xmi:type='uml:Association' or
            @xmi:type='uml:AssociationClass']">
48          <!-- look for associations -->
49          <xsl:variable name="generalID" select="generalization/@general"/>
50          <xsl:variable name="assoName" select="@name"/>
51          <xsl:variable name="assoID" select="@xmi:id"/>
52          <xsl:variable name="superName" select="//packagedElement[@xmi:id=$
                generalID]/@name"/>
53          scn_association_D(<xsl:value-of select="$assoName"/>,3,<!--
54          --><xsl:for-each select="//*[@association=$assoID]"><!--
55              --><xsl:variable name="ref" select="@xmi:id"/><!--
56              --><xsl:variable name="roleName" select="//*[@xmi:id=$ref]/
                    @name"/><!--
57              --><xsl:value-of select="//*[@xmi:id=$ref]/@name"/>,<
                    xsl:value-of select="//packagedElement[@xmi:id=//*[@xmi:id
                    =$ref]/@type]/@name"/><!--
58              --><xsl:if test="position()!=last()">,</xsl:if>
59          </xsl:for-each>).
```

If the association does not have a generalization, then it must be a top level class:

topLevel_D(<name of the **association**/**association class**>).

```
60          <xsl:if test="not($superName)">
61              topLevel_D(<xsl:value-of select="$assoName"/>).
62          </xsl:if>
```

Otherwise, if an association is a specialization of another association, then a subclass relation has to be added:

subclassOf_D(<name of the **association**/**association class**>,<name of the parent>).

[2] See CHAPTER 10.4.3 for a description of this shortcut.

```
63          <xsl:if test="$superName">
64              subclassOf_D(<xsl:value-of select="$assoName"/>,<xsl:value-of
                    select="$superName"/>).
65          </xsl:if>
```

Like above for the classes, an association class can have attributes. These are output here by the template 'translateAttribute', which is described below.

```
67          <!-- look for attributes in case its an AssociationClass -->
68          <xsl:call-template name="translateAttribute"/>
69      </xsl:template>
```

The template 'translateAttribute' is looking for <ownedAttribute> subnodes. If the attribute is typed as an enumeration, then the following code is generated:

```
hasAttr_D(<name of the class/association class>,<name of the attribute>)
    ,2,<name of the enumeration>,1,1).
```

Note that the potency is always set to 2, and for the sake of simplicity, the cardinalities are always set to a minimum and maximum of 1.

If the attribute is not typed, a string type is assumed.

```
71      <xsl:template name="translateAttribute">
72          <xsl:for-each select="ownedAttribute">
73              <xsl:variable name="type" select="@type"/>
74              <xsl:choose>
75                  <xsl:when test="$type">
76                      <xsl:variable name="attributeType" select="//
                            packagedElement[@xmi:id=$type]"/>
77                      <xsl:if test="$attributeType[@xmi:type='
                            uml:Enumeration']">
78                          hasAttr_D(<xsl:value-of select="../@name"/>,<
                                xsl:value-of select="@name"/>,2,<xsl:value-of
                                select="$attributeType/@name"/>,1,1).
79                      </xsl:if>
80                  </xsl:when>
81                  <xsl:otherwise><!-- interpret as string -->
82                      hasAttr_D(<xsl:value-of select="../@name"/>,<
                            xsl:value-of select="@name"/>,2,<xsl:value-of
                            select="@name"/>,1,1).
83                  </xsl:otherwise>
84              </xsl:choose>
85          </xsl:for-each>
86      </xsl:template>
```

The last template simply states that if any text is found while parsing the input file, it is not output directly to the result file.

```
88      <xsl:template match="text()"/>
89  </xsl:stylesheet>
```

4. Transform DML architecture package to Nivel

This section describes the script implementing the translation of a DML architecture provided as a UML model in XMI format to Nivel syntax.
The header here is the same as the one in the previous section.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
       Transform" xmlns:xmi='http://schema.omg.org/spec/XMI/2.1'>
3
4      <xsl:output method="text" encoding="ISO-8859-1"/>
5
6      <xsl:template match="/">
7          <xsl:apply-templates/>
8      </xsl:template>
```

If a package is found that is named neither 'architecture', nor 'framework' or 'helpers', then it is ignored.

```
10     <xsl:template match="packagedElement[@xmi:type='uml:Package' and @name
           !='architecture' and @name!='framework' and @name!='helpers']">
11         <!-- do nothing, since this is not architecture relevant -->
12     </xsl:template>
```

As mentioned above, the purpose of this script is to translate all classes in the DML architecture to Nivel syntax. This purpose is fulfilled by a template farther below. When this is done during translation, all classes defined in the DML architecture reside at level 2 of the Nivel model and can be instantiated on level 1 for a concrete DML. Since there are classes in the DML framework that have to be instantiated at level 1, too, like feature, there have to be representatives of these classes in level 2. This is due to the implementation of deep instantiation in the Nivel framework. These representatives are understood as implicit classes: a class in the framework implies a representative class in the architecture to be able to instantiate the class in a DML.
In order to enable the creation of framework elements in a DML, each element in the DML framework is instantiated at level 2 as an instance with the same name, but having the string '_ARCH' appended to its name. This new element can then be referred to in a Nivel DML as an element of level 2 representing an element that actually resides on level 3.
The template following this description is realizing this approach. If a package is found with the name 'framework' or 'helpers', then for all classes, associations, and association classes the following code is generated:

instanceOf_D(<name of the **class**>_ARCH,<name of the **class**>).

If such an element is involved within an association, it must be assured that all associated classes that were defined in the parent association are compatible with the subclasses. For this purpose the predicate *scn_subAssociationMultiple_D* is used. This predicate has been explained in CHAPTER 10.4.3. The predicate is inserted

whenever a class in the DML framework is related to an association. The result in such a case is:

scn_subAssociationMultiple_D(<name of the framework **association**>_ARCH,<name of the framework **association**>,<name of **association** end 1>,<name of the **class** assigned to **association** end 1>_ARCH,<name of associating end 2>,<name of the **class** assigned to **association** end 2>).

Note that the type of association end 1 is concatenated with '_ARCH'. This ensures that any implicit architecture element created by the template at hand is associated to any subclass of the class assigned to association end 2.

```
14  <xsl:template match="packagedElement[@xmi:type='uml:Package' and (
        @name='framework' or @name='helpers')]">
15      %% create implicit architecture elements from '<xsl:value-of
            select="@name"/>' %%
16      <xsl:for-each select="packagedElement[@xmi:type='uml:Class' or
            @xmi:type='uml:Association' or @xmi:type='uml:AssociationClass
            ']">
17          <xsl:variable name="classID" select="@xmi:id"/>
18          instanceOf_D(<xsl:value-of select="@name"/>_ARCH,<xsl:value-of
                select="@name"/>).
19          <xsl:variable name="generalID" select="generalization/@general
                "/>
20          <xsl:variable name="className" select="@name"/>
21          <xsl:for-each select="//*[@type=$classID]"> <!-- looking for "
                ownedAttribute" and "ownedEnd"-->
22              <xsl:variable name="thisEnd" select="."/>
23              <xsl:variable name="assoID" select="@association"/>
24              <xsl:variable name="asso" select="//packagedElement[
                    @xmi:id=$assoID]"/>
25              <xsl:variable name="otherEnd" select="//*[@xmi:id!=$
                    thisEnd/@xmi:id and @association=$assoID]"/>
26              scn_subAssociationMultiple_D(<xsl:value-of select="$asso/
                    @name"/>_ARCH,<xsl:value-of select="$asso/@name"/>,<
                    xsl:value-of select="$thisEnd/@name"/>,<xsl:value-of
                    select="//*[@xmi:id=$thisEnd/@type]/@name"/>_ARCH,<
                    xsl:value-of select="$otherEnd/@name"/>,<xsl:value-of
                    select="//*[@xmi:id=$otherEnd/@type]/@name"/>).
27          </xsl:for-each>
28      </xsl:for-each>
29  </xsl:template>
```

The next template simply completes the list of templates parsing the packages. It states that if the package 'architecture' is found, then the rest of the script is applied to its elements.

```
31  <xsl:template match="packagedElement[@xmi:type='uml:Package' and @name
        ='architecture']">
32      %% create explicit architecture elements %%
33      <xsl:apply-templates select="packagedElement"/>
34  </xsl:template>
```

If a class is found in the architecture package, make it an instance of its parent class in the DML framework:

instanceOf_D(<name of the **class**>,<name of the parent **class**>).

Then, the predicate *scn_subAssociationMultiple_D* is applied to any association end related to the class, like we have done above for implicit classes.

```
36    <xsl:template match="packagedElement[@xmi:type='uml:Class']">
37        <xsl:variable name="generalID" select="generalization/@general"/>
38        <xsl:variable name="generalName" select="//packagedElement[@xmi:id
              =$generalID]/@name"/>
39        <xsl:variable name="classID" select="@xmi:id"/>
40        <xsl:variable name="className" select="@name"/>
41        instanceOf_D(<xsl:value-of select="@name"/>,<xsl:value-of select="
              //packagedElement[@xmi:id=$generalID]/@name"/>).
42        <!-- for each association, create a subAssociation that inherits
              from the parent association -->
43        <xsl:for-each select="//*[@type=$generalID]"> <!-- looking for "
              ownedAttribute" and "ownedEnd"-->
44            <xsl:variable name="endID" select="@xmi:id"/>
45            <xsl:variable name="assoID" select="@association"/>
46            <xsl:variable name="asso" select="//packagedElement[@xmi:id=$
                  assoID]"/>
47            <xsl:variable name="assoName" select="$asso/@name"/>
48            <xsl:variable name="otherEnd" select="//*[@xmi:id!=$endID and
                  @association=$assoID]"/>
49            <xsl:variable name="otherAttribute" select="//*[@xmi:id=$
                  otherEnd/@xmi:id]"/>
50            scn_subAssociationMultiple_D(<xsl:value-of select="$assoName"/
                  >_ARCH,<xsl:value-of select="$assoName"/>,<xsl:value-of
                  select="@name"/>,<xsl:value-of select="$className"/>,<
                  xsl:value-of select="$otherAttribute/@name"/>,<xsl:value-of
                   select="//*[@xmi:id=$otherAttribute/@type]/@name"/>).<!--
                  * is looking for "ownedAttribute" and "ownedEnd"-->
51        </xsl:for-each>
52    </xsl:template>
```

For associations and association classes, the translation is quite the same as for classes, but at the end of the following template, a role playing is defined for all association ends attached to the association:

playsRoleIn_D(<**class** of the **association** end>,<name of the **association** end >,<name of the **association**>).

```
54    <xsl:template match="packagedElement[@xmi:type='uml:Association' or
          @xmi:type='uml:AssociationClass']">
55        <xsl:variable name="generalID" select="generalization/@general"/>
56        <xsl:variable name="assoName" select="@name"/>
57        <xsl:variable name="superName" select="//packagedElement[@xmi:id=$
              generalID]/@name"/>
58        instanceOf_D(<xsl:value-of select="$assoName"/>,<xsl:value-of
              select="$superName"/>).
59        <!-- for each association, create a subAssociation that inherits
              from the parent association -->
60        <xsl:for-each select="//*[@type=$generalID]"> <!-- looking for "
              ownedAttribute" and "ownedEnd"-->
```

```
61              <xsl:variable name="thisEnd" select="."/>
62              <xsl:variable name="assoID" select="@association"/>
63              <xsl:variable name="asso" select="//packagedElement[@xmi:id=$
                    assoID]"/>
64              <xsl:variable name="otherEnd" select="//*[@xmi:id!=$thisEnd/
                    @xmi:id and @association=$assoID]"/>
65              scn_subAssociationMultiple_D(<xsl:value-of select="$asso/@name
                    "/>,<xsl:value-of select="$asso/@name"/>,<xsl:value-of
                    select="$thisEnd/@name"/>,<xsl:value-of select="//*[@xmi:id
                    =$thisEnd/@type]/@name"/>,<xsl:value-of select="$otherEnd/
                    @name"/>,<xsl:value-of select="//*[@xmi:id=$otherEnd/@type
                    ]/@name"/>).
66          </xsl:for-each>
67          <xsl:for-each select="ownedEnd">
68              <xsl:variable name="ref" select="@xmi:id"/>
69              <xsl:variable name="roleName" select="//*[@xmi:id=$ref]/@name"
                    />
70              playsRoleIn_D(<xsl:value-of select="//packagedElement[@xmi:id
                    =//*[@xmi:id=$ref]/@type]/@name"/>,<xsl:value-of select="
                    //*[@xmi:id=$ref]/@name"/>,<xsl:value-of select="$assoName"
                    />).
71          </xsl:for-each>
72      </xsl:template>
73
74      <xsl:template match="text()"/>
75  </xsl:stylesheet>
```

5. Transform DML package to Nivel

Here, the XSL implementation of the translation of a DML to a UML model in XMI format to Nivel syntax is explained.
The script starts with a common XSL script header. Note that the output format is 'text'.

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
        Transform" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3
4       <xsl:output method="text" encoding="ISO-8859-1"/>
5
6       <xsl:template match="/">
7           <xsl:apply-templates/>
8       </xsl:template>
```

The first functional template is the following. It states that every package that is neither named 'dml', nor 'extension' and 'configuration' is ignored in the output.

```
10      <xsl:template match="packagedElement[@xmi:type='uml:Package' and @name
            !='dml' and @name!='extension' and @name!='configuration']">
11          <!-- do nothing, since this is not dml relevant -->
12      </xsl:template>
```

The next two templates represent the opposite of the previous template. If a package found is either named 'dml' 'extension', or 'configuration', then its contents are transfered to the output model. In case of 'extension', another *lparse* comment is placed in the output file. Comments do not have an impact on functionality, though.

```
14      <xsl:template match="packagedElement[@xmi:type='uml:Package' and (
            @name='dml' or @name='configuration'])">
15  %% creating development model line content %%
16          <xsl:apply-templates select="packagedElement"/>
17      </xsl:template>
18
19      <xsl:template match="packagedElement[@xmi:type='uml:Package' and @name
            ='extension']">
20  %% creating development model line extension content %%
21          <xsl:apply-templates select="packagedElement"/>
22      </xsl:template>
```

The main work to be done during creation of a Nivel DML model is to look out for instances. These are identified in <packagedElement> nodes with a <xmi:type> attribute value of 'uml:InstanceSpecification'. An instance must have a classifier. The output generated by the template below is the following:

instanceOf_D(<name of the instance>,<name of the classifier>).
instanceOf_d(<name of the instance>,<name of the classifier>).

An instance is here both declared (<**instanceOf_D**>) and actualized (<**instanceOf_d**>). This ensures that the instance is actual an element in the model and *smodels* will not be tempted to remove the element from the model in search of a stable model without it.
In case the instance is an instance of a DML framework element, then an '_ARCH' is added to the name of the classifier, because otherwise, the Nivel framework would identify a consistency error when finding an instance at level 1 that is directly related to another instance residing at level 3.

instanceOf_D(<name of the instance>,<name of the classifier>_ARCH).
instanceOf_d(<name of the instance>,<name of the classifier>_ARCH).

If an instance in the model is not named, then the XSL script inserts a randomly generated name, instead. This name is created using the XSL function generate-id. The parameter is the node the name is created for, assuring that the same identifier is created again at later times if the same node is passed to the function.
If we look into the details, we notice that the classifier is identified in two possible ways. It's either stored as an attribute right with the <packagedElement> node. That's the way MagicDraw represents the classifier information. Or, the classifier might be stored in a sub-node <classifier>. This is the way the Eclipse Modeling Framework stores classifiers.

```
24      <xsl:template match="packagedElement[@xmi:type='
            uml:InstanceSpecification']">
25          <xsl:variable name="classifierID">
```

```
26              <xsl:choose>
27                  <xsl:when test="@classifier"> <!-- MagicDraw uses
                        classifier attribute -->
28                      <xsl:value-of select="@classifier"/>
29                  </xsl:when>
30                  <xsl:otherwise> <!-- Eclipse Modeling Framework uses
                        classifier sub element -->
31                      <xsl:value-of select="classifier/@xmi:idref"/>
32                  </xsl:otherwise>
33              </xsl:choose>
34          </xsl:variable>
35          <xsl:variable name="classifierName" select="//packagedElement[
                @xmi:id=$classifierID]/@name"/>
36          <xsl:variable name="instance" select="."/>
37          <xsl:variable name="instanceName">
38              <xsl:choose>
39                  <xsl:when test="@name!=''"><xsl:value-of select="@name"/><
                        /xsl:when>
40                  <xsl:otherwise><xsl:value-of select="generate-id(.)"/></
                        xsl:otherwise>
41              </xsl:choose>
42          </xsl:variable>
43          <!-- only go on if there is a classifier -->
44          <xsl:if test="$classifierName">
45              <!-- if the classifier comes out of the framework, the
                    appendix "_ARCH" is added to its name! -->
46              instanceOf_D(<xsl:value-of select="$instanceName"/>,<
                    xsl:value-of select="$classifierName"/><xsl:if test="//*[
                    @xmi:id=$classifierID]/../@name='framework' or //*[@xmi:id
                    =$classifierID]/../@name='helpers'">_ARCH</xsl:if>).
47              <!-- make the instance actual! -->
48              instanceOf_d(<xsl:value-of select="$instanceName"/>,<
                    xsl:value-of select="$classifierName"/><xsl:if test="//*[
                    @xmi:id=$classifierID]/../@name='framework' or //*[@xmi:id
                    =$classifierID]/../@name='helpers'">_ARCH</xsl:if>).
49              <xsl:apply-templates select="slot">
50                  <xsl:with-param name="instanceName" select="$instanceName"
                        />
51                  <xsl:with-param name="instance" select="$instance"/>
52              </xsl:apply-templates>
53          </xsl:if>
54      </xsl:template>
```

Some instances will have associations to other instances. This information is stored in <slot> nodes lying within a <packagedElement>.
If an attribute has the type string, indicated by slot value 'uml:LiteralString' for the node <xmi:type>, then the following output is generated:

hasValue_D(<name of the instance>,<name of the **association** end's type>,<value of the slot>).
contains(<name of the association end's type>,<value of the slot>).

This adds both the slot value to the instance, as well as to the domain of possible

values for the string. This domain has to be stated in order to keep the amount of alternatives to be checked by *smodels* within finite bounds.
If the slot represents the end of an association, then the role playing is added to the model:

playsRoleIn_D(<name of the related element>,<name of the **association** end >,<name of the instance>).

If the slot represents the value of an enumeration, the following code is generated:

hasValue_D(<name of the instance>,<name of the **association** end's type>,< value of the slot>).

Here, no **contains** predicate is needed since the domain of values for the enumeration has already been spanned in the DML framework.
As above, when an instance has not been fitted with a name, the function generate-id is used. The function creates the same identifier that was created during instantiation of the element, since the function is actually returning a hash code of the element.

```
56    <xsl:template match="slot">
57        <xsl:param name="instanceName"/>
58        <xsl:param name="instance"/>
59
60        <xsl:variable name="roleID" select="@definingFeature"/>
61        <xsl:variable name="role" select="//*[@xmi:id=$roleID]"/>
62        <xsl:variable name="targetInstanceID" select="value/@instance"/>
63        <xsl:variable name="targetInstance" select="//*[@xmi:id=$
              targetInstanceID]"/>
64        <!-- only interpret this slot as role if its not a literal -->
65        <xsl:choose>
66            <xsl:when test="value/@xmi:type='uml:LiteralString'">
67                hasValue_D(<xsl:value-of select="$instanceName"/>,<
                      xsl:value-of select="$role/@name"/>,<xsl:value-of
                      select="value/@value"/>).
68                contains(<xsl:value-of select="$role/@name"/>,<
                      xsl:value-of select="value/@value"/>).
69            </xsl:when>
70            <xsl:when test="not(//ownedLiteral[@xmi:id=$targetInstanceID])
                  ">
71                <xsl:variable name="elementName">
72                    <xsl:if test="not($targetInstance/@name)">
73                        <xsl:value-of select="generate-id($targetInstance)
                              "/>
74                    </xsl:if>
75                    <xsl:if test="$targetInstance/@name">
76                        <xsl:value-of select="$targetInstance/@name"/>
77                    </xsl:if>
78                </xsl:variable>
79                playsRoleIn_D(<xsl:value-of select="$elementName"/>,<
                      xsl:value-of select="$role/@name"/>,<xsl:value-of
                      select="$instanceName"/>).
80
```

```
81              </xsl:when>
82              <xsl:otherwise>
83                  hasValue_D(<xsl:value-of select="$instanceName"/>,<
                        xsl:value-of select="$role/@name"/>,<xsl:value-of
                        select="$targetInstance/@name"/>).
84              </xsl:otherwise>
85          </xsl:choose>
86      </xsl:template>
```

The rest of the script is self-explaining, the more as it has already been explained above.

```
88      <xsl:template match="text()"/>
89
90  </xsl:stylesheet>
```

DML in Nivel syntax

6. Transformed DML framework in Nivel syntax

This section contains the complete DML framework illustrated in FIGURE 9.2 in Nivel syntax.
The code is identical to the code generated by the UML-to-Nivel transformation for the DML framework (see CHAPTER 10.4.4), but it is stripped off whitespaces and dispensable newlines.

```
scn_association_D(deselects,3,deselector,featureDeselector,
    deselectedFeature,feature).
     topLevel_D(deselects).

topLevel_D(featureDeselector).
     hasPotency_D(featureDeselector,3).

topLevel_D(feature).
     hasPotency_D(feature,3).

subclassOf_D(contentElement,modelElement).

topLevel_D(operand).
     hasPotency_D(operand,3).

scn_association_D(modifies,3,operation,changeOperation,target,modelElement
    ).
     topLevel_D(modifies).

scn_association_D(boundConstraint,3,oper,constraintOperand,constr,
    constraint).
     topLevel_D(boundConstraint).

topLevel_D(constraint).
     hasPotency_D(constraint,3).
     hasAttr_D(constraint,invalMode,2,invalidityMode,1,1).

scn_association_D(constrainedElement,3,constr,constraint,element,
    contentElement).
     topLevel_D(constrainedElement).

scn_association_D(relation,3,source,contentElement,target,contentElement).
     subclassOf_D(relation,contentElement).

contains(operatorType,opAND).
```

```
32 contains(operatorType,opOR).
33 contains(operatorType,opNOT).
34 contains(operatorType,opXOR).
35 contains(operatorType,opID).
36
37 scn_association_D(belongs_to,3,element,modelElement,capsule,feature).
38     topLevel_D(belongs_to).
39
40 subclassOf_D(rootTerm,term).
41
42 scn_association_D(operands,3,operandsContainer,term,operandsContained,
       operand).
43     topLevel_D(operands).
44
45 scn_association_D(rootFeature,3,root,feature,model,featureModel).
46     topLevel_D(rootFeature).
47
48 subclassOf_D(term,operand).
49     hasAttr_D(term,operator,2,operatorType,1,1).
50
51 contains(cardinalityType,mandatory).
52 contains(cardinalityType,optional).
53
54 scn_association_D(parameter,3,operation,changeOperation,param,modelElement
       ).
55     topLevel_D(parameter).
56
57 scn_association_D(subfeature,3,whole,feature,part,feature).
58     topLevel_D(subfeature).
59     hasAttr_D(subfeature,cardinality,2,cardinalityType,1,1).
60
61 subclassOf_D(typeConstraint,constraint).
62     hasAttr_D(typeConstraint,constrainedTypeName,2,constrainedTypeName
           ,1,1).
63
64 subclassOf_D(constraintOperand,operand).
65
66 topLevel_D(featureModel).
67     hasPotency_D(featureModel,3).
68
69 topLevel_D(modelElement).
70     hasPotency_D(modelElement,3).
71
72 contains(invalidityMode,invalid).
73 contains(invalidityMode,true).
74 contains(invalidityMode,false).
75
76 subclassOf_D(changeOperation,modelElement).
```

7. Transformed DML architecture in Nivel syntax

This section contains the complete exemplary DML architecture illustrated in FIGURE 9.2.
The code is identical to the code generated by the UML-to-Nivel transformation for DML architectures (see CHAPTER 10.4.4), but it is stripped off whitespaces and dispensable newlines.

```
1  instanceOf_D(product,contentElement).
2      scn_subAssociationMultiple_D(relation_ARCH,relation,source,product,
           target,contentElement).
3      scn_subAssociationMultiple_D(relation_ARCH,relation,target,product,
           source,contentElement).
4      scn_subAssociationMultiple_D(constrainedElement_ARCH,
           constrainedElement,element,product,constr,constraint).
5
6  instanceOf_D(modifiesResponsibility,modifies).
7      playsRoleIn_D(changeProductResponsibility,operation,
           modifiesResponsibility).
8      playsRoleIn_D(responsibility,target,modifiesResponsibility).
9
10 instanceOf_D(role,contentElement).
11     scn_subAssociationMultiple_D(relation_ARCH,relation,source,role,target
           ,contentElement).
12     scn_subAssociationMultiple_D(relation_ARCH,relation,target,role,source
           ,contentElement).
13     scn_subAssociationMultiple_D(constrainedElement_ARCH,
           constrainedElement,element,role,constr,constraint).
14
15 instanceOf_D(hasToBeResponsibleFor,constraint).
16     scn_subAssociationMultiple_D(boundConstraint_ARCH,boundConstraint,
           constr,hasToBeResponsibleFor,oper,constraintOperand).
17     scn_subAssociationMultiple_D(constrainedElement_ARCH,
           constrainedElement,constr,hasToBeResponsibleFor,element,
           contentElement).
18
19 instanceOf_D(newRole,parameter).
20     playsRoleIn_D(changeProductResponsibility,operation,newRole).
21     playsRoleIn_D(role,param,newRole).
22
23 instanceOf_D(responsibility,relation).
24     playsRoleIn_D(role,source,responsibility).
25     playsRoleIn_D(product,target,responsibility).
26
27 instanceOf_D(changeProductResponsibility,changeOperation).
28     scn_subAssociationMultiple_D(modifies_ARCH,modifies,operation,
           changeProductResponsibility,target,modelElement).
29     scn_subAssociationMultiple_D(parameter_ARCH,parameter,operation,
           changeProductResponsibility,param,modelElement).
30
31 instanceOf_D(constrainedRole,constrainedElement).
32     playsRoleIn_D(hasToBeResponsibleFor,constr,constrainedRole).
```

```
33      playsRoleIn_D(role,element,constrainedRole).
34
35  instanceOf_D(constrainedProduct,constrainedElement).
36      playsRoleIn_D(hasToBeResponsibleFor,constr,constrainedProduct).
37      playsRoleIn_D(product,element,constrainedProduct).
38
39  instanceOf_D(hasToBeRelatedToAny,typeConstraint).
40
41  instanceOf_D(deselects_ARCH,deselects).
42
43  instanceOf_D(featureDeselector_ARCH,featureDeselector).
44      scn_subAssociationMultiple_D(deselects_ARCH,deselects,deselector,
            featureDeselector_ARCH,deselectedFeature,feature).
45
46  instanceOf_D(feature_ARCH,feature).
47      scn_subAssociationMultiple_D(deselects_ARCH,deselects,
            deselectedFeature,feature_ARCH,deselector,featureDeselector).
48      scn_subAssociationMultiple_D(belongs_to_ARCH,belongs_to,capsule,
            feature_ARCH,element,modelElement).
49      scn_subAssociationMultiple_D(rootFeature_ARCH,rootFeature,root,
            feature_ARCH,model,featureModel).
50      scn_subAssociationMultiple_D(subfeature_ARCH,subfeature,whole,
            feature_ARCH,part,feature).
51      scn_subAssociationMultiple_D(subfeature_ARCH,subfeature,part,
            feature_ARCH,whole,feature).
52
53  instanceOf_D(contentElement_ARCH,contentElement).
54      scn_subAssociationMultiple_D(relation_ARCH,relation,source,
            contentElement_ARCH,target,contentElement).
55      scn_subAssociationMultiple_D(relation_ARCH,relation,target,
            contentElement_ARCH,source,contentElement).
56      scn_subAssociationMultiple_D(constrainedElement_ARCH,
            constrainedElement,element,contentElement_ARCH,constr,constraint).
57
58  instanceOf_D(operand_ARCH,operand).
59      scn_subAssociationMultiple_D(operands_ARCH,operands,operandsContained,
            operand_ARCH,operandsContainer,term).
60
61  instanceOf_D(modifies_ARCH,modifies).
62
63  instanceOf_D(boundConstraint_ARCH,boundConstraint).
64
65  instanceOf_D(constraint_ARCH,constraint).
66      scn_subAssociationMultiple_D(boundConstraint_ARCH,boundConstraint,
            constr,constraint_ARCH,oper,constraintOperand).
67      scn_subAssociationMultiple_D(constrainedElement_ARCH,
            constrainedElement,constr,constraint_ARCH,element,contentElement).
68
69  instanceOf_D(constrainedElement_ARCH,constrainedElement).
70
71  instanceOf_D(relation_ARCH,relation).
72
```

```
73 instanceOf_D(belongs_to_ARCH, belongs_to).
74
75 instanceOf_D(rootTerm_ARCH, rootTerm).
76
77 instanceOf_D(operands_ARCH, operands).
78
79 instanceOf_D(rootFeature_ARCH, rootFeature).
80
81 instanceOf_D(term_ARCH, term).
82     scn_subAssociationMultiple_D(operands_ARCH, operands, operandsContainer,
       term_ARCH, operandsContained, operand).
83
84 instanceOf_D(parameter_ARCH, parameter).
85
86 instanceOf_D(subfeature_ARCH, subfeature).
87
88 instanceOf_D(typeConstraint_ARCH, typeConstraint).
89
90 instanceOf_D(constraintOperand_ARCH, constraintOperand).
91     scn_subAssociationMultiple_D(boundConstraint_ARCH, boundConstraint, oper
       , constraintOperand_ARCH, constr, constraint).
92
93 instanceOf_D(featureModel_ARCH, featureModel).
94     scn_subAssociationMultiple_D(rootFeature_ARCH, rootFeature, model,
       featureModel_ARCH, root, feature).
95
96 instanceOf_D(modelElement_ARCH, modelElement).
97     scn_subAssociationMultiple_D(modifies_ARCH, modifies, target,
       modelElement_ARCH, operation, changeOperation).
98     scn_subAssociationMultiple_D(belongs_to_ARCH, belongs_to, element,
       modelElement_ARCH, capsule, feature).
99     scn_subAssociationMultiple_D(parameter_ARCH, parameter, param,
       modelElement_ARCH, operation, changeOperation).
100
101 instanceOf_D(changeOperation_ARCH, changeOperation).
102     scn_subAssociationMultiple_D(modifies_ARCH, modifies, operation,
       changeOperation_ARCH, target, modelElement).
103     scn_subAssociationMultiple_D(parameter_ARCH, parameter, operation,
       changeOperation_ARCH, param, modelElement).
```

8. Transformed DML in Nivel syntax

This section contains the complete exemplary DML illustrated in FIGURE 9.3, after execution of configuration and modification by the DML environment.
The code is identical to the code generated by the UML-to-Nivel transformation for DML's (see CHAPTER 10.4.4), but it is stripped off whitespaces and dispensable newlines.

```
1 instanceOf_D(subterm1,term_ARCH).
2 instanceOf_d(subterm1,term_ARCH).
```

```
3       hasValue_D(subterm1, operator, opXOR).
4
5   instanceOf_D(d1e206, belongs_to_ARCH).
6   instanceOf_d(d1e206, belongs_to_ARCH).
7       playsRoleIn_D(pmResp4pm, element, d1e206).
8       playsRoleIn_D(projManagement, capsule, d1e206).
9
10  instanceOf_D(d1e219, belongs_to_ARCH).
11  instanceOf_d(d1e219, belongs_to_ARCH).
12      playsRoleIn_D(projManagement, capsule, d1e219).
13      playsRoleIn_D(projectManual, element, d1e219).
14
15  instanceOf_D(d1e232, boundConstraint_ARCH).
16  instanceOf_d(d1e232, boundConstraint_ARCH).
17      playsRoleIn_D(qaHasToBeResponsible4qaManual, constr, d1e232).
18      playsRoleIn_D(operand3, oper, d1e232).
19
20  instanceOf_D(d1e245, operands_ARCH).
21  instanceOf_d(d1e245, operands_ARCH).
22      playsRoleIn_D(mainManagementProducts, operandsContainer, d1e245).
23      playsRoleIn_D(subterm1, operandsContained, d1e245).
24
25  instanceOf_D(pmrepo, featureModel_ARCH).
26  instanceOf_d(pmrepo, featureModel_ARCH).
27
28  instanceOf_D(projectManager, role).
29  instanceOf_d(projectManager, role).
30
31  instanceOf_D(qaHasToBeRelated2anyProduct, hasToBeRelatedToAny).
32  instanceOf_d(qaHasToBeRelated2anyProduct, hasToBeRelatedToAny).
33      hasValue_D(qaHasToBeRelated2anyProduct, constrainedTypeName, product).
34      contains(constrainedTypeName, product).
35      hasValue_D(qaHasToBeRelated2anyProduct, invalMode, false).
36
37  instanceOf_D(operand1, constraintOperand_ARCH).
38  instanceOf_d(operand1, constraintOperand_ARCH).
39
40  instanceOf_D(pmHasToBeResponsible4pmManual, hasToBeResponsibleFor).
41  instanceOf_d(pmHasToBeResponsible4pmManual, hasToBeResponsibleFor).
42      hasValue_D(pmHasToBeResponsible4pmManual, invalMode, invalid).
43
44  instanceOf_D(qaHasToBeResponsible4qaManual, hasToBeResponsibleFor).
45  instanceOf_d(qaHasToBeResponsible4qaManual, hasToBeResponsibleFor).
46      hasValue_D(qaHasToBeResponsible4qaManual, invalMode, false).
47
48  instanceOf_D(d1e295, boundConstraint_ARCH).
49  instanceOf_d(d1e295, boundConstraint_ARCH).
50      playsRoleIn_D(operand1, oper, d1e295).
51      playsRoleIn_D(pmHasToBeResponsible4pmManual, constr, d1e295).
52
53  instanceOf_D(projectManual, product).
54  instanceOf_d(projectManual, product).
```

```
55
56  instanceOf_D(subterm2,term_ARCH).
57  instanceOf_d(subterm2,term_ARCH).
58      hasValue_D(subterm2,operator,opNOT).
59
60  instanceOf_D(d1e318,belongs_to_ARCH).
61  instanceOf_d(d1e318,belongs_to_ARCH).
62      playsRoleIn_D(projManagement,capsule,d1e318).
63      playsRoleIn_D(projectManager,element,d1e318).
64
65  instanceOf_D(d1e331,operands_ARCH).
66  instanceOf_d(d1e331,operands_ARCH).
67      playsRoleIn_D(subterm1,operandsContainer,d1e331).
68      playsRoleIn_D(subterm2,operandsContained,d1e331).
69
70  instanceOf_D(mainManagementProducts,rootTerm_ARCH).
71  instanceOf_d(mainManagementProducts,rootTerm_ARCH).
72      hasValue_D(mainManagementProducts,operator,opAND).
73
74  instanceOf_D(d1e468,boundConstraint_ARCH).
75  instanceOf_d(d1e468,boundConstraint_ARCH).
76      playsRoleIn_D(operand2,oper,d1e468).
77      playsRoleIn_D(qaHasToBeRelated2anyProduct,constr,d1e468).
78
79  instanceOf_D(orgModel,feature_ARCH).
80  instanceOf_d(orgModel,feature_ARCH).
81
82  instanceOf_D(d1e483,rootFeature_ARCH).
83  instanceOf_d(d1e483,rootFeature_ARCH).
84      playsRoleIn_D(orgModel,root,d1e483).
85      playsRoleIn_D(pmrepo,model,d1e483).
86
87  instanceOf_D(d1e496,subfeature_ARCH).
88  instanceOf_d(d1e496,subfeature_ARCH).
89      hasValue_D(d1e496,cardinality,mandatory).
90      playsRoleIn_D(orgModel,whole,d1e496).
91      playsRoleIn_D(projManagement,part,d1e496).
92
93  instanceOf_D(operand3,constraintOperand_ARCH).
94  instanceOf_d(operand3,constraintOperand_ARCH).
95
96  instanceOf_D(d1e517,constrainedRole).
97  instanceOf_d(d1e517,constrainedRole).
98      playsRoleIn_D(projectManager,element,d1e517).
99      playsRoleIn_D(pmHasToBeResponsible4pmManual,constr,d1e517).
100
101 instanceOf_D(d1e530,constrainedProduct).
102 instanceOf_d(d1e530,constrainedProduct).
103     playsRoleIn_D(pmHasToBeResponsible4pmManual,constr,d1e530).
104     playsRoleIn_D(projectManual,element,d1e530).
105
106 instanceOf_D(pmResp4pm,responsibility).
```

```
107 instanceOf_d(pmResp4pm, responsibility).
108     playsRoleIn_D(projectManager, source, pmResp4pm).
109     playsRoleIn_D(projectManual, target, pmResp4pm).
110
111 instanceOf_D(d1e556, operands_ARCH).
112 instanceOf_d(d1e556, operands_ARCH).
113     playsRoleIn_D(operand3, operandsContained, d1e556).
114     playsRoleIn_D(subterm1, operandsContainer, d1e556).
115
116 instanceOf_D(d1e569, operands_ARCH).
117 instanceOf_d(d1e569, operands_ARCH).
118     playsRoleIn_D(operand2, operandsContained, d1e569).
119     playsRoleIn_D(subterm2, operandsContainer, d1e569).
120
121 instanceOf_D(projManagement, feature_ARCH).
122 instanceOf_d(projManagement, feature_ARCH).
123
124 instanceOf_D(operand2, constraintOperand_ARCH).
125 instanceOf_d(operand2, constraintOperand_ARCH).
126
127 instanceOf_D(d1e587, operands_ARCH).
128 instanceOf_d(d1e587, operands_ARCH).
129     playsRoleIn_D(mainManagementProducts, operandsContainer, d1e587).
130     playsRoleIn_D(operand1, operandsContained, d1e587).
```

DML semantics

9. Semantics for constraint restriction

This section covers the semantics for constraint restriction embedded in the DML environment. It contains all rules concerning constraints, without the definition of the constraintIsFulfilled predicate. This predicate is part of the DML architecture, while the following rules belong to the DML framework and must not be altered during DML architecture creation.
The code can be divided into several sections:

- **Shortcuts and Helpers (lines 1ff.)**: These are used in the other sections.
- **Spanning variant room(lines 48ff.)**: For each $\bar{o}$perand in any $\bar{t}$erm, one of two states are initially assumed. It may be either satisfied or unsatisfied. This code section creates two model variants for each $\bar{o}$perand. In accordance to additional rules defined below, none, one, or both variants may be discarded from the set of stable models.
- **Semantics for constraint synchronization (lines 57ff.)**: The complete spanning of variant space is synchronized with the actual satisfaction status of $\bar{o}$perands.
- **Assembly of terms using boolean operators (lines 75ff.)**: implementation of boolean logics for the combination of $\bar{o}$perands to $\bar{t}$erms.
- **Semantics of the attribute $\bar{i}$nvalMode (lines 157ff.)**: In case of a configured and reduced DML model, some $\bar{c}$onstraints may point to targets that were removed from the model. The semantics of $\bar{i}$nvalMode specify how to evaluate such a constraint.

The following code is documented inline. With the short descriptions above, and the exact specification in the code, it should be possible to get the meaning. In some cases, further descriptions are provided inline.

```
1 %% SHORTCUTS AND HELPERS %%
2 %% Shortcuts %%
3 % Shortcut for association "boundConstraint" instances between
      BoundConstraint and constraint
4 operand2constraint(Operand, Constraint) :- instanceOf_tp(Constraint,
      constraint,2),
5     instanceOf_tp(Operand, constraintOperand,2),
6     instanceOf_tp(BoundConstraint, boundConstraint,2),
```

```
7       playsRoleIn_D(Constraint,constr,BoundConstraint),
8       playsRoleIn_D(Operand,oper,BoundConstraint).
9
10  % Shortcut for association "constrainedElement" instances between an
        constraint and a contentElement
11  constrainedElement(Constraint,Element) :- instanceOf_tp(Element,
        contentElement,2),
12      instanceOf_tp(Constraint,constraint,2),
13      instanceOf_tp(Asso,constrainedElement,2),
14      playsRoleIn_D(Element,element,Asso),
15      playsRoleIn_D(Constraint,constr,Asso).
16
17  % Shortcut for the type specified by a typeConstraint
18  constrainedType(Constraint,Type) :- instanceOf_tp(Type,contentElement,1),
19      instanceOf_tp(Constraint,typeConstraint,2),
20      hasValue_D(Constraint,constrainedType,Type).
21
22  % Shortcuts for access to operands of terms
23  isOperandOf(Operand,Term) :- instanceOf_tp(Operands,operands,2),
24      instanceOf_tp(Operand,operand,2),
25      instanceOf_tp(Term,term,2),
26      playsRoleIn_D(Term,operandsContainer,Operands),
27      playsRoleIn_D(Operand,operandsContained,Operands).
28  areOperandsOf(Operand,Operand2,Term) :- instanceOf_tp(Operand,operand,2),
29      instanceOf_tp(Operand2,operand,2),
30      instanceOf_tp(Term,term,2),
31      Operand != Operand2,
32      isOperandOf(Operand,Term),
33      isOperandOf(Operand2,Term).
34
35  % Shortcuts for operatorTypes
36  opTypeAND(Term) :- instanceOf_tp(Term,term,2),
37      hasValue_D(Term,operator,opAND).
38  opTypeNOT(Term) :- instanceOf_tp(Term,term,2),
39      hasValue_D(Term,operator,opNOT).
40  opTypeID(Term) :- instanceOf_tp(Term,term,2),
41      hasValue_D(Term,operator,opID).
42  opTypeOR(Term) :- instanceOf_tp(Term,term,2),
43      hasValue_D(Term,operator,opOR).
44  opTypeXOR(Term) :- instanceOf_tp(Term,term,2),
45      hasValue_D(Term,operator,opXOR).
```

The following rule represents the implementation of Requirement 1 we know from CHAPTER 10.4.6.

```
48  %% SPAN VARIANTS %%
49  % An operand may either be satisfied, or unsatisfied, never both!
50  1 { satisfiedOperand(Operand), unsatisfiedOperand(Operand) } 1 :-
        instanceOf_tp(Operand,operand,2).
```

This next rule implements Requirement 3 from CHAPTER 10.4.6.

```
52  % A root operand that is unsatisfied disqualifies the model containing it!
53   :- unsatisfiedOperand(RootTerm),
```

```
54      instanceOf_tp(RootTerm,rootTerm,2).
```

The following rules assure that if a constraint is identified as satisfied, then the model variant where it is marked as unsatisfied is discarded, and vice versa.

```
57 %% DEFINE SEMANTICS FRAMEWORK FOR CONSTRAINTS %%
58 % Discard all models were constraintIsFulfilled and satisfiedOperand are
       unsynchronized for a constraintOperand/constraint pair
59  :- 1 { constraintIsFulfilled(Constraint), satisfiedOperand(Operand) } 1,
60      operand2constraint(Operand,Constraint),
61      instanceOf_tp(Operand,constraintOperand,2),
62      instanceOf_tp(Constraint,constraint,2).
63
64 % Discard all models were checkUnsatisfiedOperand and unsatisfiedOperand
       are unsynchronized
65  :- 1 { constraintIsNotFulfilled(Constraint), unsatisfiedOperand(Operand)
       } 1,
66      operand2constraint(Operand,Constraint),
67      instanceOf_tp(Operand,constraintOperand,2),
68      instanceOf_tp(Constraint,constraint,2).
69
70 % If the semantics provided in the architecture does not infer the
       predicate constraintIsFulfilled for a constraint, then the constraint
       is not fulfilled.
71 constraintIsNotFulfilled(Constraint) :- not constraintIsFulfilled(
       Constraint),
72      instanceOf_tp(Constraint,constraint,2).
```

The operator logics implement Requirement 2 from CHAPTER 10.4.6. It discards all models were any of the boolean logics is not set correctly. For example, think of a term x with two operands a and b that has an operatorType of opAND.
Remember that the rule in line 50 creates four possible models for each assignment of satisfiedOperand and unsatisfiedOperand to a and b. term x implies with its associated AND logics that all models have to be discarded that do not have both the predicates satisfiedOperand(a) and satisfiedOperand(b) set. Thus, three models will be regarded as invalid that contain the following predicates:

1. unsatisfiedOperand(a). satisfiedOperand(b).
2. satisfiedOperand(a). unsatisfiedOperand(b).
3. unsatisfiedOperand(a). unsatisfiedOperand(b).

```
75 %% DEFINE LOGICS FOR OPERATORS %%
76 %% opAND %%
77 % the model is invalid if term is satisfied, and any operand is not
       satisfied
78  :- instanceOf_tp(Term,term,2),
79      instanceOf_tp(Operand,operand,2),
80      opTypeAND(Term),
81      satisfiedOperand(Term),
82      isOperandOf(Operand,Term),
```

```
83      not satisfiedOperand(Operand).
84  % the model is invalid if term is unsatisfied, and both operands are
        satisfied
85   :- instanceOf_tp(Term,term,2),
86      instanceOf_tp(Operand,operand,2),
87      instanceOf_tp(Operand2,operand,2),
88      opTypeAND(Term),
89      not satisfiedOperand(Term),
90      areOperandsOf(Operand,Operand2,Term),
91      satisfiedOperand(Operand),
92      satisfiedOperand(Operand2).
93
94  %% opOR %%
95  % the model is invalid if term is satisfied, and both operands are not
96   :- instanceOf_tp(Term,term,2),
97      instanceOf_tp(Operand,operand,2),
98      instanceOf_tp(Operand2,operand,2),
99      opTypeOR(Term),
100     satisfiedOperand(Term),
101     areOperandsOf(Operand,Operand2,Term),
102     not satisfiedOperand(Operand),
103     not satisfiedOperand(Operand2).
104 % the model is invalid if term is unsatisfied, and any operand is
        satisfied
105  :- instanceOf_tp(Term,term,2),
106     instanceOf_tp(Operand,operand,2),
107     opTypeOR(Term),
108     not satisfiedOperand(Term),
109     isOperandOf(Operand,Term),
110     satisfiedOperand(Operand).
111
112 %% opXOR %%
113 % the model is invalid if term is satisfied, and both operand are
        satisfied
114  :- instanceOf_tp(Term,term,2),
115     instanceOf_tp(Operand,operand,2),
116     instanceOf_tp(Operand2,operand,2),
117     opTypeXOR(Term),
118     satisfiedOperand(Term),
119     areOperandsOf(Operand,Operand2,Term),
120     satisfiedOperand(Operand),
121     satisfiedOperand(Operand2).
122 % the model is invalid if term is satisfied, and both operand are not
        satisfied
123  :- instanceOf_tp(Term,term,2),
124     instanceOf_tp(Operand,operand,2),
125     instanceOf_tp(Operand2,operand,2),
126     opTypeXOR(Term),
127     satisfiedOperand(Term),
128     areOperandsOf(Operand,Operand2,Term),
129     not satisfiedOperand(Operand),
130     not satisfiedOperand(Operand2).
```

```
131 % the model is invalid if term is unsatisfied, and exactly one operand is
        satisfied
132  :- 1 { satisfiedOperand(Operand), satisfiedOperand(Operand2) } 1,
133     instanceOf_tp(Term,term,2),
134     instanceOf_tp(Operand,operand,2),
135     instanceOf_tp(Operand2,operand,2),
136     opTypeXOR(Term),
137     not satisfiedOperand(Term),
138     areOperandsOf(Operand,Operand2,Term).
139
140 %% opNOT %%
141 % the model is invalid, if the term is satisfied and the operand is, too,
        or if both are not
142  :- 1 { satisfiedOperand(Term), not satisfiedOperand(Operand) } 1,
143     instanceOf_tp(Term,term,2),
144     instanceOf_tp(Operand,operand,2),
145     opTypeNOT(Term),
146     isOperandOf(Operand,Term).
147
148 %% opID %%
149 % the model is invalid, if the term is satisfied and the operand isn't, or
        vice versa
150  :- 1 { satisfiedOperand(Term), satisfiedOperand(Operand) } 1,
151     instanceOf_tp(Term,term,2),
152     instanceOf_tp(Operand,operand,2),
153     opTypeID(Term),
154     isOperandOf(Operand,Term).
```

The semantics of constraint's attribute invalMode is shown below. The comments at the beginning of the following code describe in natural language how to handle the three different values invalid, true, and false.

```
157 %% SEMANTICS OF invalMode %%
158 %% Check if a constraint that ought to have targets does not have a target
        ; %%
159 % Background: after reduction of the model, it may be that the target of a
160 % constraint may have been removed from the model. If this is the case, it
161 % has to be stated how this constraint is to be interpreted. A constraint
162 % has an attribute "invalMode" that can have three values:
163 % a) invalid - the whole term using this constraint must be regarded as
        evaluating to false
164 % b) true - the constraint is regarded as evaluating to true
165 % c) false - the constraint is regarded as evaluating to false
166 constraintHasTarget(Constraint) :-
167     instanceOf_tp(Constraint,constraint,2),
168     instanceOf_tp(Target,contentElement,2),
169     constrainedElement(Constraint,Element).
170
171 % See constraint as unsatisfied, if invalMode is false and no targets
172 constraintIsNotFulfilled(Constraint) :- not constraintHasTarget(Constraint
        ),
173     instanceOf_tp(Constraint,constraint,2),
174     hasValue_D(Constraint,invalMode,false).
```

```
175
176 % See constraint as satisfied, if invalMode is false and no targets
177 constraintIsFulfilled(Constraint) :- not constraintHasTarget(Constraint),
178     instanceOf_tp(Constraint,constraint,2),
179     hasValue_D(Constraint,invalMode,true).
180
181 % A constraint is regarded invalid, if invalMode is 'invalid'.
182 invalidConstraint(Constraint) :- not constraintHasTarget(Constraint),
183     instanceOf_tp(Constraint,constraint,2),
184     hasValue_D(Constraint,invalMode,invalid).
185
186 invalidOperand(Operand) :- operand2constraint(Operand,Constraint),
187     instanceOf_tp(Constraint,constraint,2),
188     instanceOf_tp(Operand,constraintOperand,2),
189     invalidConstraint(Constraint).
190
191 % Any term using an invalid operand is itself invalid
192 invalidOperand(Term) :- isOperandOf(Operand,Term),
193     instanceOf_tp(Operand,operand,2),
194     instanceOf_tp(Term,term,2),
195     invalidOperand(Operand).
196
197 % Any invalid Operand is unsatisfied
198 unsatisfiedOperand(Operand) :- invalidOperand(Operand),
199     instanceOf_tp(Operand,operand,2).
```

Bibliography and indices

Bibliography

[AEH+08] Ove Armbrust, Jan Ebell, Ulrike Hammerschall, Jürgen Münch, and Daniela Thoma. Experiences and results from tailoring and deploying a large process standard in a company. *Software Process: Improvement and Practice*, 13(4):301–309, John Wiley & Sons, Chichester, Great Britain, July 2008.

[AF98] Paul Allen and Stuart Frost. *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*. Cambridge University Press, 1998.

[AG01] Michalis Anastasopoulos and Cristina Gacek. Implementing product line variabilities. *ACM SIGSOFT Software Engineering Notes*, 26(3):109–117, ACM, New York, USA, 2001.

[AK01] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 19–33. Springer-Verlag, 2001.

[AK02] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22, 2002.

[AKM+08] Ove Armbrust, Masafumi Katahira, Yuko Miyamoto, Jürgen Münch, Haruka Nakao, and Alexis Ocampo. Scoping software process models - initial concepts and experience from defining space standards. In *Proceedings of the International Conference on Software Process, 2008*, volume 5007 of *LNCS*, pages 160 – 172, Springer, Berlin/Heidelberg, Germany, May 2008.

[AKM+09] Ove Armbrust, Masafumi Katahira, Yuko Miyamoto, Jürgen Münch, Haruka Nakao, and Alexis Ocampo. Scoping software process lines. *Software Process: Improvement and Practice*, 14(3):181–197, John Wiley & Sons, New York, USA, 2009.

[AM09] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4):521–549, 2009.

[AMS06] Timo Asikainen, Tomi Mannisto, and Timo Soininen. A unified conceptual foundation for feature modelling. In *Proceedings of the 10th International Software Product Line Conference, 2006*, pages 31–40, IEEE Computer Society, Washington, USA, 2006.

[ANS04] ANSI American National Standards Institute. Ansi/eia-649-a 2004: National consensus standard for configuration management, 2004.

[Arm08] Ove Armbrust. Leitfaden zur Modelleinführung im Rahmen der organisations-spezifischen Anpassung des V-Modell XT. Technical Report 013.08/D, Institut für Experimentelles Software Engineering (IESE), March 2008.

[Asi07] Timo Asikainen. Nivel on the soberit software business and engineering institute webpage, version made public on 2007-8-29. `http://www.soberit.tkk.fi/nivel/`, 2007.

[Asi08] Timo Asikainen. *A conceptual modeling approach to software variability*. PhD thesis, Helsinki University of Technology, 2008.

[ASU92] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau*, volume 1. Addison-Wesley, Bonn, 2nd edition, 1992.

[Atk97] Colin Atkinson. Meta-Modeling for distributed object environments. In *Proceedings of the 1st International Conference on Enterprise Distributed Object Computing, EDOC '97*, pages 90–101, IEEE Computer Society, Washington, DC, USA, 1997.

[BBM05] Kathrin Berg, Judith Bishop, and Dirk Muthig. Tracing software product line variability: from problem to solution space. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 182–191, South African Institute for Computer Scientists and Information Technologists, White River, South Africa, 2005.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. `http://agilemanifesto.org/`, 2001. Access: 8th January 2010.

[BCG83] Robert Balzer, Thomas E. Cheatham, Jr., and Cordell Green. Software technology in the 1990's: Using a new paradigm. *Computer*, 16(11):39–45, 1983.

[Ben87] Herbert D. Benington. Production of large computer programs. In *Proceedings of the 9th international conference on Software Engineering*, pages 299–310, IEEE Computer Society Press, Monterey, CA, USA, 1987.

[BF01] Kent Beck and Martin Fowler. *Planning extreme programming*. Addison-Wesley, 2001.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering, 2001*, LNCS, pages 303–338, Springer, Berlin/Heidelberg, Germany, 2002.

[BFT09] Christian Bartelt, Edward Fischer, and Thomas Ternité. Paradigmen zur Variabilitätsbeschreibung von Vorgehensmodellen. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *INFORMATIK 2009 – Im Focus das Leben*, volume P-154 of *GI-Lecture Notes in Informatics*, pages 3507–3521, Bonner Köllen, Bonn, September 2009.

[BMI97] BMI Bundesministerium des Innern. V-Model [97] – development standard for it-systems of the federal republic of germany, lifecycle process model, brief description. `http://v-modell.iabg.de/index.php?option=com_docman&task=doc_details&gid=13&Itemid=30`, 1997. Access: 12th January 2010.

[BMI08] BMI Bundesministerium des Innern. V-Model XT 1.3 HTML documentation. `http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.3/V-Modell%20XT%20HTML%20English/`, 2008. Access: 13th January 2010.

[BMI10] BMI Bundesministerium des Innern. Das V-Modell XT. `http://www.v-modell-xt.de`, 2010. Access: 13th January 2010.

[BMMB97] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Begtsson. Object-oriented frameworks: Problems & experiences. Research Report HKR-RES-97/9-SE, University of Karlskrona/Ronneby, Ronneby, Sweden, 1997.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, Chichester, Great Britain, 1st edition, 1996.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, IEEE Computer Society, Los Alamitos,, May 1988.

[BPS04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, Elsevier, Amsterdam, Netherlands, 2004.

[BR05] Manfred Broy and Andreas Rausch. Das neue V-Modell XT: Ein anpassbares Modell für Software und Systems Engineering. *Informatik Spektrum*, 28(3):220–229, Springer, Berlin/Heidelberg, Germany, June 2005.

[CKK06] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International Software Product Line Conference, SPLC 2006*, pages 41–51, IEEE Computer Society, Washington, USA, 2006.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, August 2002.

[Coc93] Alistair A. R. Cockburn. The impact of object-orientation on application development. *IBM Systems Journal*, 32(3):420–444, 1993.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[DIN01] DIN Deutsches Institut für Normung. DIN 69901: Projektmanagement; Projektmanagementsysteme; . . . , 2001.

[dOGHM05] Edson Alves de Oliveira, Jr., Itana M. S. Gimenes, Elisa Hatsue Moriya Huzita, and José Carlos Maldonado. A variability management process for software product lines. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research, 2005*, pages 225–241. IBM Press, 2005.

[DSB09] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Variability assessment in software product families. *Information and Software Technology*, 51(1):195–218, Butterworth-Heinemann, Newton, USA, 2009.

[Ecl10a] Eclipse. ATL, atlas transformation language, version 3.1.0. `http://www.eclipse.org/atl/`, June 2010. Access: 20th July 2010.

[Ecl10b] Eclipse. Eclipse modeling framework. `http://www.eclipse.org/emf`, 2010.

[EV10] J. Laurenz Eveleens and Chris Verhoef. The rise and fall of the Chaos Report figures. *IEEE Software*, 27(1):30–36, 2010.

[FK07] Martin Fritzsche and Patrick Keil. Kategorisierung etablierter vorgehensmodelle und ihre verbreitung in der deutschen Software-Industrie. Technical Report TUM-I0717, Technische Universität München, 2007.

[FM92] Kevin Forsberg and Harold Mooz. The relationship of systems engineering to the project cycle. *Engineering Managament Journal*, 4(3):36–43, 1992.

[FM01] Kevin Forsberg and Harold Mooz. A visual explanation of development methods and strategies including the waterfall, spiral, vee, vee+, vee++ models. In *Proceedings of the International Council for Systems Engineering Conference, INCOSE 2001*, `http://www.csm.com/Repository/Model/rep/o/pdf/VisualExplanationofModels.pdf`, Melbourne, Australia, 2001.

[FR05] Rainer Frömming and Andreas Rausch. Migration of persistent object models using XMI. In Hongji Yang, editor, *Advances in UML and XML Based Software Evolution*, pages 92–104. Idea Group Publishing, Jul 2005.

[FW06] Joyce Fortune and Diana White. Framing of project critical success factors by a systems model. *International Journal of Project Management*, 24(1):53–65, 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Amsterdam, Netherlands, 1st edition, 1994.

[Gib97] J. Paul Gibson. Feature requirements models: Understanding interactions. In *Proceedings of the IEEE Fourth International Workshop on Feature Interactions in Networks and Distributed Systems, FIW'97*, pages 46–60, IOS Press, Amsterdam, Netherlands, 1997.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming, 1988*, pages 1070–1080. MIT Press, 1988.

[Gla06] Robert L. Glass. The Standish report: does it really describe a software crisis? *Commun. ACM*, 49(8):15–16, 2006.

[Gna07] Michael Andreas Josef Gnatz. *Vom Vorgehensmodell zum Projektplan*. PhD thesis, Technische Universität München, München, Germany, December 2007.

[Gol09] Golimojo. Make object-oriented programming easier with only six lines of javascript. `http://www.golimojo.com/etc/js-subclass.html`, 2009. Access: 13th August 2009.

[HHMS04] Bernd Hindel, Klaus Hörmann, Markus Müller, and Jürgen Schmied. *Basiswissen Software-Projektmanagement*. Dpunkt.Verlag GmbH, 1st edition, June 2004.

[HK10a] Jonas Helming and Maximilian Kögel. Unicase. `http://unicase.org`, 2010.

[HK10b] Markus Herrmannsdörfer and Maximilian Kögel. Towards a generic operation recorder for model evolution. In *Proceedings of the International Workshop on Model Comparison in Practice, IWMCP'2010*, pages 76 – 81, ACM, New York, USA, July 2010.

[IEE98] IEEE Computer Society. IEEE 828-1998: IEEE standard for software configuration management plans, 1998. ISBN 0-7381-0331-4.

[ISB03] ISB Informatikstrategieorgan Bund. HERMES – Führen und Abwickeln von Projekten der Informations- und Kommunikationstechnik (IKT) – Grundwissen. `http://www.hermes.admin.ch/ikt_projektfuehrung/handbuecher/handbucher-als-pdf-zum-download/hermes-grundwissen/at_download/file`, 2003. Access: 13th January 2010.

[ISB09] ISB Informatikstrategieorgan Bund. Die Entwicklung von HERMES. `http://www.hermes.admin.ch/ueber-hermes/projekte?set_language=de&cl=de`, 2009. Access: 13th January 2010.

[ISO95] ISO International Organisation for Standardization. ISO/IEC 12207: Software life cycle processes. `http://www.abelia.com/docs/12207cpt.pdf`, 1995.

[ISO00] ISO International Organisation for Standardization. ISO 9001: Quality management systems requirements. `http://www.iso9001.qmb.info/`, 2000. Access: 7th January 2010.

[ISO03] ISO International Organisation for Standardization. ISO/IEC 10007:2003: Quality management systems – guidelines for configuration management, 2003.

[ISO06] ISO International Organisation for Standardization. ISO/IEC 15504: Spice (software process improvement and capability determination), 2006.

[ISO09] ISO International Organisation for Standardization. ISO 31000: Risk management – principles and guidelines on implementation, 2009.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1st edition, 1999.

[Joh91] Ralph E. Johnson. Reusing object-oriented design. Technical Report UIUCDCS 91-1696, University of Illinois, Illinois, USA, 1991.

[Joh02] Rod Johnson. *Expert One-on-One J2EE Design and Development*. Wrox Press, 2002.

[JS02] Lawrence G. Jones and Albert L. Soule. Software process improvement and product line practice: Capability Maturity Model Integration (CMMI) and the framework for software product line practice. Technical Report CMU/SEI-2002-TN-012, Software Engineering Institute, July 2002.

[KCH+90] Kyo Kang, Sholom Cohen, James A. Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[KH10a] Maximilian Kögel and Jonas Helming. Emfstore. `http://emfstore.org`, 2010.

[KH10b] Maximilian Kögel and Jonas Helming. EMFStore - a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, Track on Information Research Demos*, pages 307–308, ACM, New York, USA, 2010.

[KL02] Arun Kishan and Monica Lam. Dynamic kernel modification and extensibility. Technical report, Stanford University, Stanford, 2002.

[KN05] Marco Kuhrmann and Dirk Niebuhr. Das V-Modell XT in der Praxis - IT-WiBe. In Roland Petrasch, Reinhard Höhn, Stephan Höppner, Herbert Wetzel, and Manuela Wiemers, editors, *Proceedings of the 12. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI), Entscheidungsfall Vorgehensmodelle, 2005*, pages 81–92, Shaker, Aachen, Germany, 2005.

[Kna04] Peter Knauber. Managing the evolution of software product lines. In *Proceedings of the 8th International Conference on Software Reuse, ICSR-8, 2004*, LNCS, Springer, Berlin/Heidelberg, Germany, 2004.

[KR01] Frank Keienburg and Andreas Rausch. Using XML/XMI for tool supported evolution of UML models. In *Proceedings of 34th annual Hawaii international conference on system sciences, HICSS'34*, pages 3–6. IEEE Computer Society, Jan 2001.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2nd edition, 2000.

[KS07] Thomas Kühne and Daniel Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepjava. *ACM SIGPLAN Notices*, 42(10):229–244, October 2007.

[KT09] Marco Kuhrmann and Thomas Ternité. Das V-Modell XT 1.3 - Neuerungen für Anwender und Prozessingenieure. In Reinhard Höhn and Oliver Linssen, editors, *Proceedings of the 16. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI), Vorgehensmodelle und Implementierungsfragen Akquisition - Lokalisierung - soziale Manahmen - Werkzeuge*, pages 97–108, Shaker, Aachen, Germany, April 2009.

[Kuh05] Marco Kuhrmann. Projektspezifische Anpassungen nach dem Tailoring des V-Modell XT durchführen. In Hubert Biskup and Ralf Kneuper, editors, *Proceedings of the 13. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI), Nutzen und Nutzung von Vorgehensmodellen, 2006*, pages 27–42, Shaker, Aachen, Germany, 2005.

[Kuh08] Marco Kuhrmann. *Konstruktion modularer Vorgehensmodelle*. PhD thesis, Technische Universität München, July 2008.

[LK04] Kwanwoo Lee and Kyo C. Kang. *Feature Dependency Analysis for Product Line Component Design*, pages 69–85. LNCS. Springer, Berlin/Heidelberg, Germany, 2004.

[LMT07] Shirley Lacy, Ivor MacFarlane, and Sharon Taylor. *ITIL – Service Transition*. TSO The Stationary Office, 2007.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, ACM, New York, USA, 1994.

[Mär05] Friedhelm Märsch. *EJBs und J2EE - Enterprise-Anwendungen konzipieren & programmieren*. W3L, 2005.

[MC99] Luis Mandel and María Cengarle. On the expressive power of OCL. In *Proceedings of the International Conference on Formal Methods, FM'99*, volume 1708 of *LNCS*, pages 854–874, Springer, Berlin/Heidelberg, Germany, 1999.

[McG04] John D. McGregor. Software product lines. *Journal of Object Technology*, 3(3):65–74, 2004.

[MJ82] Daniel D. McCracken and Michael A. Jackson. Life cycle concept considered harmful. *SIGSOFT Softw. Eng. Notes*, 7(2):29–32, 1982.

[MT99] Victor W. Marek and Mirosław Truszcyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirek Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398. Springer, 1999.

[Nat07] National ITS Architecture Team. Systems engineering for intelligent transportation systems. Technical Report FHWA-HOP-07-069, Department of Transportation, Office of Operations, Washington, DC, USA, 2007.

[Nie99] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3):241–273, November 1999.

[Nob00] James Noble. Basic relationship patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 73–94. Addison-Wesley Longman, Amsterdam, Netherlands, 2000.

[NSS99] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning, 1999*, volume 1730 of *LNCS*, pages 317–331, Springer, London, UK, 1999.

[NSS00] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, 2000*, April 2000.

[OBM05] Alexis Ocampo, Fabio Bella, and Jürgen Münch. Software process commonality analysis. *Software Process: Improvement and Practice, Special Issue on the 5th International Workshop on Software Process Simulation and Modeling, ProSim 2004*, 10(3):273–285, John Wiley & Sons, Chichester, Great Britain, August 2005.

[Ode94] James Odell. Power types. *Journal of Object-Oriented Programming*, 7(2):8–12, May 1994.

[OGC02] OGC Office of Government Commerce, editor. *Managing Successful Projects with PRINCE2*. TSO The Stationary Office, 2002.

[OGC09a] OGC Office of Government Commerce. PRINCE2 – background. `http://www.ogc.gov.uk/methods_prince_2__background.asp`, 2009. Access: 13th January 2010.

[OGC09b] OGC Office of Government Commerce. PRojects IN Controlled Environments Home. `http://www.prince-officialsite.com/home/home.asp`, 2009. Access: 13th January 2010.

[OM09] Alexis Ocampo and Jürgen Münch. Rationale modeling for software process evolution. *Softw. Process*, 14(2):85–105, 2009.

[OMG06] OMG Object Management Group. Meta Object Facility (MOF) Core Specification, version 2.0. `http://www.omg.org/spec/MOF/2.0/`, January 2006. Access: 10th November 2009.

[OMG07a] OMG Object Management Group. MOF 2.0/XMI mapping, version 2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm, December 2007. Access: 25th May 2010.

[OMG07b] OMG Object Management Group. UML 2.1.1, omg unified modeling language (omg uml), superstructure, version 2.1.1. http://schema.omg.org/spec/UML/2.1.1/, August 2007. Access: 30th July 2010.

[OMG08a] OMG Object Management Group. QVT, MOF 2.0 query / views / transformations, version 1.0. http://www.omg.org/spec/QVT/1.0/, April 2008. Access: 20th July 2010.

[OMG08b] OMG Object Management Group. Software Process Engineering Meta-Model (SPEM), version 2.0. http://www.omg.org/technology/documents/formal/spem.htm, April 2008. Access: 19th June 2009.

[OMG10a] OMG Object Management Group. OCL 2.2, object constraint language. http://www.omg.org/spec/OCL/2.2/, February 2010. Access: 12th May 2010.

[OMG10b] OMG Object Management Group. UML 2.3, omg unified modeling language (omg uml), superstructure, version 2.3. http://schema.omg.org/spec/UML/2.3/, May 2010. Access: 24th June 2010.

[OMR09] Alexis Ocampo, Jürgen Münch, and William E. Riddle. Incrementally introducing process model rationale support in an organization. In *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes, ICSP 2009*, volume 5543 of *LNCS*, pages 330–341. Springer, 2009.

[OS07] Alexis Ocampo and Martín Soto. Connecting the rationale for changes to the evolution of a process. In *Prodeedings of the International Conference on Product-Focused Software Process Improvement, PROFES 2007*, volume 4589/2007 of *LNCS*, pages 160–174. Springer, 2007.

[PF02] Stephen R. Palmer and John M. Felsing. *A Practical Guide to Feature Driven Development*. Prentice Hall, 2002.

[Pit93] Matthew Pittman. Lessons learned in managing object-oriented development. *IEEE Software*, 10(1):43–53, 1993.

[PSU09a] PSU Pennsylvania State University. CiteseerX website. http://citeseerx.ist.psu.edu, 2009. Access: 19th June 2009.

[PSU09b] PSU Pennsylvania State University. CiteseerX website: Search results for "feature-oriented domain analysis". http://citeseerx.ist.psu.edu/search?q=title%3A(Feature-Oriented+Domain+Analysis)+AND+year%3A1990&sort=cite&ic=1, 2009. Access: 19th June 2009.

[PvdLM06] Klaus Pohl, Frank van der Linden, and Andreas Metzger. Software product line variability management. In *Proceedings of the 10th International on Software Product Line Conference, 2006*, page 219, IEEE Computer Society, Washington, USA, 2006.

[RK08] Maryam Razavian and Ramtin Khosravi. Modeling variability in business process models using UML. In *Proceedings of the 5th International Conference on Information Technology: New Generations, ITNG 2008*, pages 82–87, IEEE Computer Society, Washington, USA, 2008.

[Rom05] Dieter Rombach. Integrated software process and product lines. In Mingshu Li, Barry Boehm, and Leon J. Osterweil, editors, *Unifying the Software Process Spectrum*, volume 3840 of *LNCS*, pages 83 – 90. Springer, Berlin/Heidelberg, Germany, May 2005.

[Roy87] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338, IEEE Computer Society Press, Monterey, CA, USA, 1987.

[Sam01] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, Berlin/Heidelberg, Germany, 1st edition, 2001.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[Sch06] Arnd Schnieders. Variability mechanism centric process family architectures. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS 2006*, pages 289–298, IEEE Computer Society, Washington, USA, 2006.

[SCO07] Borislava Simidchieva, Lori Clarke, and Leon Osterweil. Representing process variation with a process family. In *Proceedings of the International Conference on Software Process, ICSP 2007*, LNCS, pages 109–120, Springer, Berlin/Heidelberg, Germany, 2007.

[SEI06] SEI Software Engineering Institute. Capability Maturity Model Integration. `http://www.sei.cmu.edu/cmmi/`, 2006. Access: 19th June 2009.

[Smu00] Raymond M. Smullyan. *Forever Undecided*. Oxford Paperbacks, November 2000.

[SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[SP06] Arnd Schnieders and Frank Puhlmann. Variability mechanisms in E-Business process families. In *Proceedings of the 9th International Conference on Business Information Systems, BIS 2006*, volume P-85 of *GI-Lecture Notes in Informatics*, pages 583–601, Bonner Köllen, Bonn, Germany, 2006.

[Spr09] SpringSource. Official Spring website. `http://www.springsource.org/`, 2009. Access: 19th June 2009.

[Sta94] Standish Group International. The CHAOS report, reprint. `http://www.projectsmart.co.uk/docs/chaos-report.pdf`, 1994. Access: 4th January 2010.

[Sun09] Sun Microsystems. Official Java EE website. `http://java.sun.com/javaee/`, 2009. Access: 19th June 2009.

[SvGB02] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software Practice & Experience*, 35(8):705—754, John Wiley & Sons, Chichester, Great Britain, 2002.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, ACM, New York, USA, 1996.

[TC06] Gabriele Taentzer and Giovanni Toffetti Carughi. A Graph-Based approach to transform XML documents. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering, FASE 2006*, volume 3922 of *LNCS*. Springer, 2006.

[Ter09] Thomas Ternité. Process lines: a product line approach designed for process model development. In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications, SPPI Track, SEAA 2009*, pages 173–180, IEEE Computer Society, Washington, USA, 2009.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture, WICSA'01*, pages 45–54. IEEE Computer Society, 2001.

[VMX09] VMXT Project Team. V-Modell XT Editor and V-Modell XT Projektassistent. `http://fourever.sourceforge.net/`, 2009. Access: 04th August 2010.

[W3C99] W3C World Wide Web Consortium. XSL transformations (XSLT), version 1.0. `http://www.w3.org/TR/xslt`, November 1999. Access: 20th July 2010.

[Was06] Hironori Washizaki. Building software process line architectures from bottom up. In *Proceedings of the 7th International Conference, PROFES 2006*, volume 4034 of *LNCS*, pages 415–421. Springer, Berlin/Heidelberg, Germany, 2006.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, ACM, New York, USA, 1990.

[Wik10] Wikipedia Community. IBM rational unified process. `http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process`, 2010. Access: 13th January 2010.

[Wis06] Alexander Wise. Little-JIL 1.5 language report. Technical Report UM-CS-2006-51, University of Massachusetts, Amherst, USA, 2006.

[WZ88] Peter Wegner and Stanley Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP 1988*, LNCS, pages 55–77, Springer, Berlin/Heidelberg, Germany, 1988.

[Zav93] Pamela Zave. Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29, IEEE Computer Society, Los Alamitos, USA, 1993.

[ZTSJ09] Liming Zhu, Tu Tran, Mark Staples, and Ross Jeffery. Technical software development process in the XML domain. In *Proceedings of the International Conference on Software Process, ICSP 2009*, volume 5543 of *LNCS*, pages 246–255, Springer, Berlin/Heidelberg, Germany, 2009.

Index of DML framework entities

This is a list of all elements of the DML framework that were discussed in the thesis. Note that not all association ends were described in the text. Therefore, some may not appear in this index.

R

S

T

Index (content)