Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Quality Driven Model Transformations for Feedback Provisioning

Doctoral Dissertation of:
Mauro Luigi Drago
(738575)

Advisor:
Prof. Carlo Ghezzi
Co-advisor:
Prof. Raffaela Mirandola
Tutor:
Prof. Gianpaolo Cugola
Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2011 – XXIV

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I-20133 — Milano

Acknowledgements

First i have to thank with all my hearth my sweet (and future wife) Silvia, i have no word to say how important she was, is, and will be for me and my mental sanity. Then come my relatives: my daddy Pasquale, my mommy Maria, my sister Francesca (all the best for your future), my grandparents Franco and Pina, my grandparents Mauro Luigi (the captain) and Maria, all my uncles and cousins.

Then i have to thank my advisor Carlo (one of the most energetic and clever persons i have ever met), my co-advisor Raffaela (source of guidance in the most obscure periods of my Ph.D.), and Judith (for letting me discover Microsoft$^{\text{tm}}$).

And now we get to the sanitarium... Here a special mention must first go to all the friends that accompanied me during the Ph.D. (the order of the list is completely random, as it came to my mind): the "Network God" (*whose presence made possible the completion of my Ph.D.*), Luca (*MPLM - he knows what this means*), Liliana (*la marchesa*), Daniel (*the leader of "Stupidiario*), Giordano (*il paperizzatore*), Andrea (*MAM, he knows what this means*), Antonio (*the word spitter*), Marco (*il Telegattone*), Matteo (*the oracle*), Leandro (*proud owner of the rubber band gun*), Andrea (*the nerd*), Alessandro (*the inventor of the fundamental law of homosexuality*), Alessandro (*il saldafusibili*), Amir (*aka technobar and the inventor of international Italian*), Marcello (*SAT-Solver g(a/u)y*), Achille (*liberate nos (ex Inferis | ab Achille)*), Mario (*Dubino seren(dip)?ity*), Santo (*the boss of the "mbari" that does everything with OneCard^{tm}*), Nicola (*thank you for letting me occupy your desk with my junk*), Diego (*and his strange Spanish alcohol*), Paola (*the voice*), Valerio (*international expert about the difference among arancini from Palermo Vs Catania Vs Messina*), Alfredo (*Zgighet*).

Special thanks go also to the *Ghisi* group and the amazing days spent on rock walls and ice falls: Luca (*the inventor of the Ghisoflex trademark*), Giorgino (*il mangia-spit odia calcare, thank you for the rope saving my life on Oceano Irrazionale*), Tod (*for her wonderful and huge dinners*).

Abstract

Verifying that a software system has certain non-functional properties is a primary concern in many engineering fields. Although several model-driven approaches exist to predict the quality attributes of a system from design models, they still lack the proper level of automation envisioned by Model Driven Software Development. In particular, when a potential issue concerning the non-functional properties is discovered in system models, the identification of a solution is still entirely up to the engineer and to his/her experience. Automation for the interpretation of the analysis results, for the identification of the potential quality-related issues, and for the identification of the possible solutions and design alternatives is mandatory for the successful application of Model Driven Engineering techniques in the current development practice. This problem, known in literature with the term *feedback provisioning*, is the problem on which our research concentrates. In this thesis we present QVT-Rational, our multi-modeling and programmable framework to automate the detection-solution loop characteristic of feedback provisioning. QVT-Rational proposes the adoption of quality-driven model transformations to specify how alternative system variants exhibiting certain quality properties can be automatically generated and proposed as feedback to the engineer. Our framework represents a valid improvement in the process of designing for quality. It provides a language to define the non-functional properties of interest for a particular engineering domain, a language to specify requirements about them, a convenient mechanism to integrate existing analysis tool-chains in order to predict quality, and provides an engine to automatically explore the solution space and identify valid system alternatives.

Riassunto

La verifica delle proprietà non funzionali esibite da un sistema software è un aspetto di estrema importanza in molti ambiti ingegneristici. Sebbene diversi approcci esistano per predire la qualità di un sistema tramite i modelli di design, questi non forniscono ancora il livello di automazione ipotizzato dalle tecniche di sviluppo software basate sui modelli. In particolare, quando un potenziale problema riguardante le proprietà non funzionali è identificato nei modelli di design di un sistema, l'ingegnere è obbligato a fare affidamento solo sulle sue capacità e sulle sue esperienze passate per identificare una soluzione. Automazione nell'interpretazione dei risultati delle analisi di qualità, nell'evidenziazione dei potenziali problemi riguardanti le proprietà non funzionali, e nell'identificazione delle possibili soluzioni e alternative di design sono caratteristiche necessarie per il successo delle tecniche basate sui modelli nella pratica corrente dello sviluppo software. Questo problema, noto in letteratura con il termine *feedback provisioning*, è il problema sul quale si concentra la nostra ricerca. In questa tesi presentiamo QVT-Rational, il sistema multi-modello e programmibile che proponiamo per automatizzare il ciclo di identificazione-risoluzione caratteristico del *feedback provisioning*. QVT-Rational propone l'adozione di un particolare tipo di trasformazioni tra modelli, le trasformazioni tra modelli dirette dagli aspetti qualitativi, per specificare come possibili alternative di design che esibiscono determinate e diverse proprietà non funzionali possano essere automaticamente generate e proposte all'ingegnere durante le fasi di design di un software. Il nostro approccio migliora il processo di gestione delle proprietÃă non funzionali durante le fasi di progettazione. QVT-Rational fornisce infatti un linguaggio per definire le proprietà non funzionali di interesse per un particolare ambito ingegneristico, un linguaggio per specificare requisiti concernenti gli aspetti qualitativi, un meccanismo conveniente per integrare gli strumenti di analisi esistenti per predire le proprietà non funzionali di sistema a partire dai sui modelli, e un motore per esplorare in modo automatico lo spazio delle soluzioni e identificare automaticamente varianti di sistema valide rispetto ai requisiti ingegneristici.

Contents

List of Figures

List of Tables

1. Introduction

Verifying that a software system exhibits certain non-functional properties is a primary concern in many application areas. Two very different examples are embedded systems and Web-based applications, where the limited computation resources and the possible large number of users may pose serious engineering problems, respectively. Anticipating the discovery of potential issues concerning the non-functional characteristics of a system, before it is implemented, is crucial for the success of the development process and for cost mitigation.

In this direction, *model-based quality prediction* techniques hold a lot of promise. System models may be used to verify certain relevant properties of the system being developed — such as performance, reliability, or schedulability — and prevent defects discoverable only after an implementation is available. Several approaches have been proposed in literature to perform model-based quality prediction [1, 2, 3, 4, 5, 6]. However, despite the advances in this research area, the current status of the available methodologies is far from an ideal situation. Current methodologies perform well in the discovery of potential issues, but lack adequate support when it comes to interpretation of results and identification of solutions. These two tasks are usually left entirely up on the engineers, who have to rely on their individual skills and experience.

Developing high-quality software systems is however complex. The experience to identify solutions to quality-related issues is hard to achieve: it is domain specific, requires a lot of time, and few experts possess it. Methodologies to formalize and share this knowledge so that also non-experienced engineers are able to cope with non-functional concerns should be provided by modern Model Driven Software Development (MDSD) environments. This challenging problem — on which our research concentrates — is known as *feedback provisioning*: how to propose solutions to non-experienced engineers and guide them in the selection of an appropriate one when issues concerning quality attributes are detected. The kind of feedback to provide and the way to provide it depend however on the adopted methodology, and some approaches have been already proposed in literature. Examples are *rule-based* approaches [7, 8, 9], *meta-heuristic* approaches [10, 11, 12], and *Design Space Exploration (DSE)* frameworks [13, 14, 15].

Rule-based methodologies rely on a set of domain specific predefined rules to identify potential quality-related problems and to suggest modifications to the system models. These approaches, however, present several drawbacks: human intervention is required, every approach defines its own language to specify rules, and rules propose solutions only for simple issues and at the level of quality prediction models (i.e., manual intervention is required to translate the suggested changes to the abstraction level of design models). Meta-heuristic approaches leverage instead specific algorithms to explore the alternatives space and to propose solutions. Although the implementations of these approaches are rather efficient, the price to pay for this is high: implementations are usually optimized for the specific domains, quality metrics, and exploration directions for which heuristics where thought. Extensions to new metrics and directions may be thus difficult, may require knowing the details of the implementation, and may lead to inefficiencies. DSE works similarly to meta-heuristic approaches, but the alternatives space is explored by encoding the problem as a Constraint Satisfaction Problem (CSP). Although DSE approaches are extremely efficient, they suffer from the same kind of problems outlined for meta-heuristic techniques.

In this thesis we describe QVT-Rational, our customizable multi-modeling framework to tackle the feedback provisioning problem. Our approach is a rule-based approach, and we propose the adoption of model transformations and, specifically, of *quality-driven model transformations* [16, 17, 18, 19, 20] to specify the feedback rules and to concretely generate design alternatives exhibiting certain non-functional properties. Quality-driven model transformations extend model transformation languages with constructs to promote non-functional attributes to first class citizens that drive the execution of the transformation. We extend two standard model transformation languages, QVT-Operational and QVT-Relations, with the appropriate constructs to specify quality concerns, and we extend their execution semantics in order to generate the viable system variants exhibiting the required level of quality.

1.1. Contribution of This Thesis

The QVT-Rational approach described in this thesis addresses the feedback provisioning problem from the perspective of all the parties involved in the process. It provides a convenient model to define quality properties, a language to define non-functional requirements, and provides an engine to automatically explore solutions and provide guidance to engineers. The advantages of QVT-Rational with respect to the existing

approaches are several:

- **Language Uniformity:** by using widely-adopted model transformation languages to define solutions to quality issues, domain experts may reuse their knowledge and do not need to master new approaches and specific languages.
- **Quality Metrics Support:** our approach is not limited to specific metrics.
- **Environments and Tool-chains Reuse:** existing modeling environment and quality prediction tools may be plugged in as-is into our framework.
- **Automation:** the availability of an automatic, requirements-driven, exploration engine can bypass the engineer in the feedback loop.
- **High Abstraction Level:** feedback is generated and presented at the abstraction level at which the engineer works. End users are not required to know all the details of the underlying quality prediction methodologies.

1.2. Structure of This Thesis

This thesis is organized as follows.

Chapter 2 provides a brief description of the setting in which we operate, outlines the feedback provisioning problem, and describes the most important available approaches, their advantages, as well their disadvantages.

Chapter 3 describes instead the main characteristics of feedback provisioning, outlines the general idea behind our solution, and introduces the case study we use throughout this thesis to describe QVT-Rational.

Chapters 4 and 5 describe how feedback can be specified by means of QVT-based quality-driven model transformations and how these augmented transformations are concretely executed in order to generate feedback (i.e., alternative system variants), respectively.

Chapter 6 provides a brief overview of the technologies adopted for the implementation of our framework and demonstrates how it is able to provide feedback also in non-trivial modeling situations, by showing the outcome of two case studies we have run.

Chapter 7 concludes the thesis by presenting final remarks and future research ideas.

2. Problem Setting and Related Work

In this chapter we motivate our research by introducing what Model Driven Engineering (MDE) is, by outlining the *feedback provisioning* problem, and by describing the current state-of-the-art in this research field. This chapter is organized as follows.

We start first with a general discussion about Model Driven Engineering (MDE), the development setting in which we operate. MDE is a broad term encompassing several research areas whose detailed description would require an entire book by itself and would be out of scope for this thesis. As a consequence, in the following we concentrate only on the aspects relevant to the work described in this thesis, that is, we talk about the evolution of MDE in the previous years, we pinpoint its goals, and we argument about the most important methodologies that have been developed in the MDE scene.

Next, we introduce two important and emerging research areas on which our research concentrates: *model driven quality prediction*, which proposes the adoption of models to early assess the non-functional properties of software and systems, and *feedback provisioning*, a problem which arises as a consequence of quality prediction and that deals with guiding engineers in the selection among different system designs with different quality properties.

Finally, we provide a detailed description of some of the approaches existing in literature for the feedback provisioning problem, by categorizing them into macro-classes and by pinpointing the advantages and the drawbacks of each proposed solution.

2.1. Introduction to Model Driven Engineering

Model Driven Engineering (MDE) is an emerging approach for software and system development envisioning a shift in the focus from source code to models. Although source code is a model of a software system — a low level model but still a model — its low abstraction level makes impractical if not impossible many software engineering tasks. For example, tracking inconsistencies between requirements and implementation,

checking the compliance between source code and the software specification, or verifying that a software artifact exhibits certain non-functional properties is not practical when source code is the only up-to-date and available artifact.

The idea behind MDE is to represent the various facets of a software system with models at different abstraction levels, possibly in a precise and formal manner[1], and to let models live along the entire design/development process. Models should not be *throw-away* artifacts, but they should be kept up-to-date and consistent along the entire software lifecycle. The goals of MDE are several: reducing the development costs, mitigating failure risks, simplifying the engineering of complex applications and, ultimately, generating the source code of the application.

MDE is however a broad term encompassing several proposals and techniques developed in the past ten years to manage software development around the concept of modeling. Some examples are Model Driven Architecture (MDA) and Model Driven Software Development (MDSD). In the following sections we briefly describe them, we show the relationship between them, and we highlight the purpose for which they have been thought.

2.1.1. Model Driven Architecture

One of the first structured approaches proposing the adoption of models for software and system development is Model Driven Architecture (MDA) [24]. MDA has been proposed in 2000 by the Object Management Group (OMG) with the intent of bridging the gap between applications and software platforms/technologies. On one hand, the continuous evolution of platforms kicks in several advantages in terms of new business opportunities and of software development. On the other hand, this continuous evolution also represent a big challenge for development teams and companies in terms of cost, time, and money. Preserving the investments of companies through this evolution of platforms is the goal of MDA.

As a practical example, let us consider the evolution of middleware to build distributed applications. In the past years several technologies have been proposed, starting with CORBA [25] — a middleware to ease communication among the several components of a distributed application that is still used at present time by many companies — continu-

[1] We use the conditional since being precise is one of the goals of MDE, but the current practice is far from this ideal situation. See for example the various *hidden* semantic variation points [21, 22] present in the Unified Modeling Language (UML) specification [23].

ing with Java Enterprise Edition (JavaEE) [26] — a middleware taking care both of communication and of other aspects of distribution such as persistence and transactions — and ending with Web Services — a replacement of proprietary standards to let distributed applications talk with well-known and standardized protocols. Each of these technologies has its own advantages and passing from one to the other, for example to let legacy code inter-operate with new components or to accommodate the customers' needs, is easier said than done. The aforementioned technologies rely on very specific protocols and features, each technology may have several implementations requiring specific customizations (see for example the various implementations of JavaEE containers), and switching from one solution to the other requires a non-negligible effort to convert the existing code and to learn the new platform.

As we mentioned before, limiting the switching cost and maximizing reuse of existing software artifacts are the main goals of MDA. To achieve them, the idea proposed by MDA is to leverage models and, specifically, to decouple aspects concerning the software itself from aspects concerning how the software is implemented for a specific platform. This idea is well depicted in Figure 2.1. MDA proposes the adoption of two very distinct kinds of models: Platform Independent Models (PIMs) which capture all the aspects of a software system that are independent from the particular underlying platform that will be used for the implementation, and Platform Specific Models (PSMs) which reify the software system (i.e., PIMs) in the context of a particular infrastructure. The advantage of this two-layered architecture is that developers work only at the high abstraction level of PIMs and that, by specifying additional information (such as platform specific configurations) it should be possible to automatically derive one or several PSMs.

If using abstractions and separating platform specific concerns are the first two cardinal points of MDA, *automation* is the last and most important one for the MDA initiative. Automation of the transformations from PIMs to PSMs is a mandatory requirement in order to mitigate development costs, favor reuse, and achieve all the goals that MDA proposes. Concretely, automation means that such transformations from PIMs to PSMs should be carried out by computer-based tools, which encapsulate the rationale to bridge the semantic gap between these two kinds of models.

However, it has been already pointed out in literature that MDA has failed to accomplish many of its goals and has failed to find broad application in industry. As Bezivin points out in [27], one of the most important causes of the MDA failure concerns automation. At the time that the MDA proposal was thought, it was still not clear how automa-

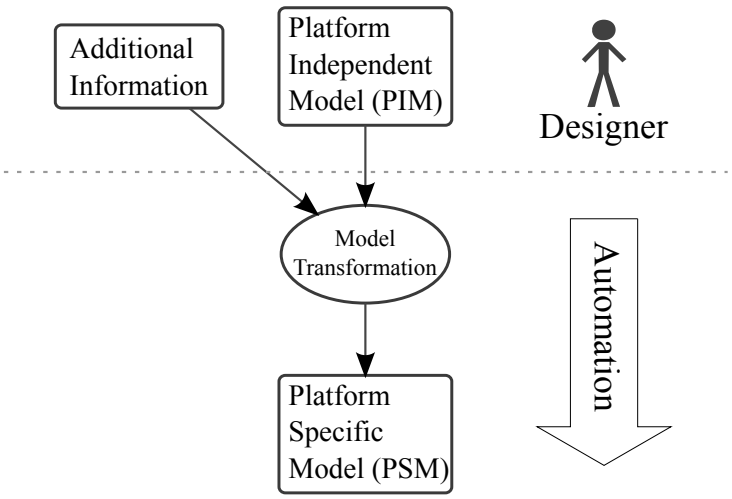

Figure 2.1.: The MDA process.

tion should be performed, i.e., how the model transformations from PIMs to PSMs should be performed, if general purpose languages or specific ones should be used to specify them, and how they should be actually executed. Another problem was that the theoretical framework on which all model-driven techniques rely (e.g., the concepts of models, meta-models, meta-meta-models, their relationship and semantics) where still not completely set. Nonetheless, MDA should be recognized the merit of bringing model-driven technologies to the attention of the great audience and of researchers. Much of the work that has been carried out in the past ten years is in fact based on the problems identified in MDA, and led to the definition of what is now known under the terms Model Driven Engineering (MDE) and Model Driven Software Development (MDSD).

2.1.2. Model Driven Engineering

As we anticipated, the failure of the MDA initiative has not brought only negative consequences. The ideas and the concepts introduced by MDA are at the foundations of what is now called Model Driven Engineering (MDE), a broader methodology for engineering encompassing all the aspects of design and development which heavily relies on models.

Several possible definitions of what MDE is exist in literature [28][29] [30][31][27]. Germane to all these definitions is the concept that MDE should not be tailored only to the PIM/PSM dichotomy, but models must encompass every aspect of software development and that MDE is not only about models, but also about automation to improve productivity

and about tools to provide a seamless experience while working with models.

Among the various definition, in this thesis we stick to the definition proposed by Bezivin in [27] for its clarity and completeness. A first important aspect highlighted by Bezivin is that MDE should not be confused with software design and development, but every engineering process (from software, to hardware, to mechanics, to physics...) which makes a heavy usage of models is a MDE process. In this sense, the relationship between MDE and software is well represented in Figure 2.2, where MDE is the general approach, MDSD refers to the usage of models to design and develop software, and MDA is an instantiation of MDSD to the specific problem of translating PIMs to PSMs.

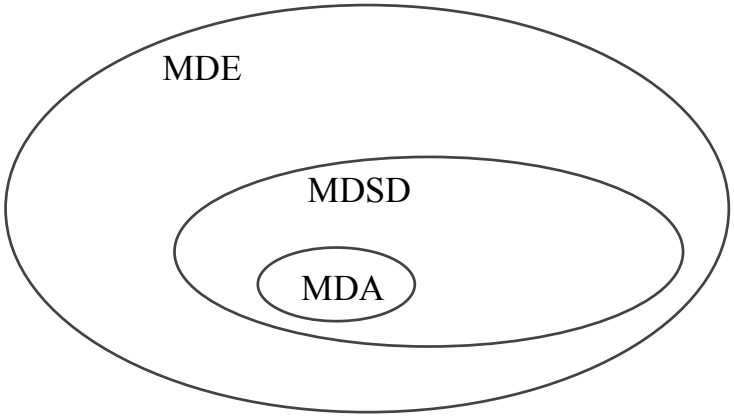

Figure 2.2.: The relationship between MDE, MDSD, and MDA.

The second important aspect of the definition of MDE to which we stick is the accent is posed on tools and automation, two important building blocks to bring models to the great audience. Principles such as the modeling stack, model transformations, or validation are important to set up a formal foundation for using models in an engineering process, but also what comes next (e.g., the standards, the tools, the languages, the technical spaces) is important. Nonetheless, the principles must be implemented in the form of practical platforms to be useful. In this sense, the whole MDE initiative can be represented by the layered architecture depicted in Figure 2.3.

On top of the methodology reside the principles on which MDE relies and which are germane to every model-based setting. An example of these well known also outside the modeling community is the concept of modeling stack. All the most important modeling initiatives available at present time share the same underlying theoretical framework for defin-

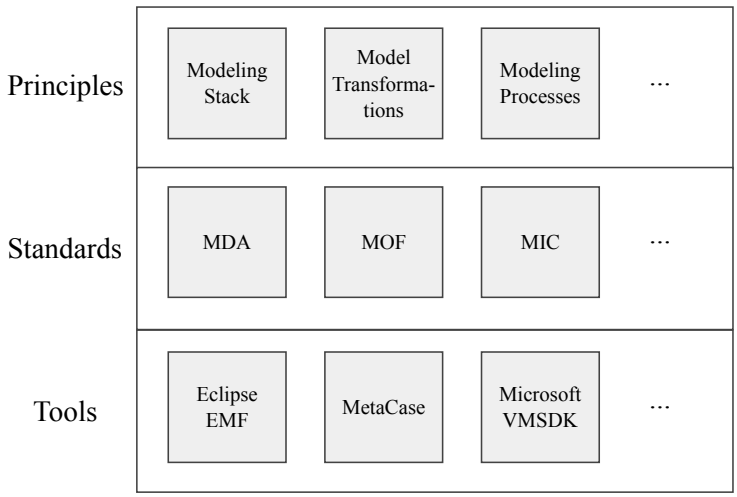

Figure 2.3.: The MDE constituents.

ing what a model is and, in detail, it is the three layered architecture represented in Figure 2.4. The *meta-meta-model* layer is self-defined and specifies the notions (or the building blocks) available to define modeling concepts. The *meta-model* layer is defined according to the concepts provided by the upper layer (to which it must conform) and declares the modeling concepts necessary for specific engineering domains. While the last layer (i.e., the *model* layer) conforms to the meta-model layer and represents a real system with the concepts provided by the meta-modeling layer. Another example of an important principle for MDE is the concept of model transformation, i.e., the idea that the modeling process can be though as a continuous application of abstraction/refinement steps which could be partially automated. Given the importance of model transformations with respect to this thesis, we will cover in detail this aspect in the next sections.

On the middle of Figure 2.2 resides instead the standards layer. Standards reify the MDE principles and are important to provide a widely-agreed framework on which tools and modeling platforms can be built. A lot of work in this direction has been carried out in the past year, and many proposals have been produced. Examples are the OMG MetaObject Facility (MOF) [32] or the Eclipse Modeling Framework (EMF) [33].

On the bottom layer resides the tools layer, or the reification of standards and principles into usable frameworks to actuate model-based design and development processes. In the previous years a lot of work has been carried out in this direction both by academia and by industries. For example, a broad selection of tools providing the necessary instru-

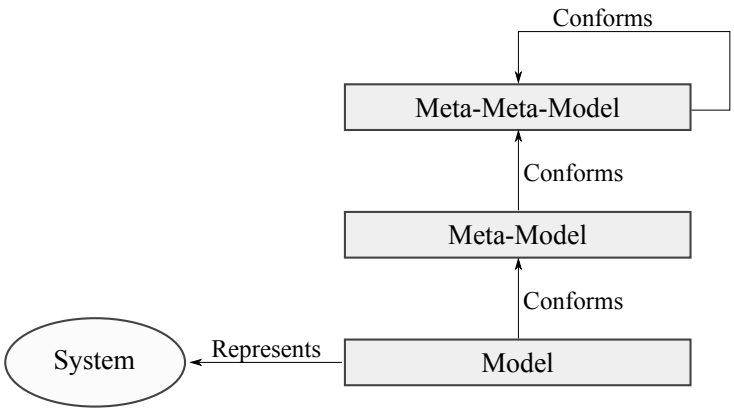

Figure 2.4.: The modeling stack constituents.

ments to create models and meta-models, to create graphical representations of them, to validate them, and to define and execute model transformations are now available on the market. Well known examples are the EMF framework for the Eclipse platform [34], the Microsoft Visual Studio Modeling and Visualization SDK (VMSDK) [35], or the tool-suite provided by Itemis a.g. to engineer Domain Specific Languages (DSLs) [36].

2.1.3. Model Transformations

Model transformations hold a fundamental role in the MDE vision. They represent the preferred mechanism to automate the design/development process and they have been recognized as *"the heart and soul of Model Driven Software Development"* [37]. Moreover, as it will be clear in the next chapters, model transformations are also the means with which we define how to generate system variants exhibiting certain quality properties, and how we propose solutions to quality-related issues.

From a conceptual point of view, several definitions of what a model transformation is have been proposed in literature. For example, the MDA proposal [24] defines a model transformation *"as the process of converting one model to another model of the same system"*. Model transformations have also been defined as models themselves as pointed out by Bezivin et al. in [38], or as *"definitions of the engineering process"* [39]. In the context of this thesis, we stick however to the much more precise definition provided by Kleppe et al. in [40]: *"a model transformation is the automatic generation of a target model from a source model, according to a transformation definition"*.

From a conceptual point of view, a model transformation may thus be

represented by the pattern depicted in Figure 2.5. To perform a model transformation both the source models and a transformation definition — the rules which determine how source model elements are actually translated into output model entities — are needed, while the output is constituted by the set of produced models as specified by the transformation definition.

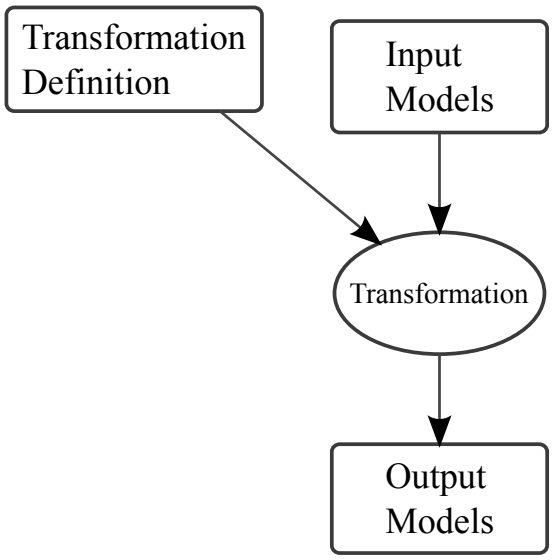

Figure 2.5.: The model transformation pattern.

In the whole process, the most important role is hold by the *transformation definition*, a fact pin-pointed also by the definition of model transformation provided by Kleppe. In the previous years, a large body of work has been done in this area and several different languages have been proposed to specify transformations. Some example are the Query View Transform (QVT) [41] family of languages proposed by the OMG, Kermeta [42], Tefkat [43], Triple Graph Grammars (TGGs) [44], and ATL [45]. Czarnecki and Helsen in [46] and Gardner et al. in [47] also provide a review of the existing languages. In the next part of this section, we outline some of the most important dimensions which characterize model transformation languages; the interested reader may find a detailed comparison of the features provided by each language and of their advantages/disadvantages in [46] and in [47].

Specification Style

A first characteristic distinguishing model transformation languages is the style proposed to define how input models map to output models.

Two main specification styles have been proposed, the *imperative* style and the *declarative* style.

The imperative specification style proposes a syntax and an execution semantics similar to widely-adopted general purpose programming languages (e.g., C# or Java), and for this reason they are also the most popular among MDE practitioners. Model transformations specified with such a style define the explicit sequence of actions to be executed in order to perform the transformation, and the languages usually provide all the standard constructs available for that purpose (e.g., loops, conditional branches, switches). Examples of languages in this class are the QVT-Operational [41] language or Kermeta [42].

The *declarative* specification style lays instead on the opposite side. Transformations specified according to such style do not define how to execute a transformation, but they define instead the relationships that must be satisfied by the elements in the input and in the output models of the transformation. It is then responsibility of the execution engine translating those relationships into an executable list of actions and performing them to produce the output models. Examples of declarative languages are QVT-Relations [41] and TGGs [44]. Declarative languages, given their syntax and execution semantics very different from the languages widely known by designers and programmers, and given the difficulty of expressing complex transformations, are popular especially in academy.

In the middle of these two opposites there is also a intermediate category of languages called *hybrid* languages, which exhibit both imperative and declarative characteristics. Hybrid languages usually promote the usage of relationships to define how input model entities map to output model entities, but they enable the specification of such relationships both in a declarative manner and in an imperative manner. This is the case for example of the ATL [45] language, or of the QVT family of languages if considered as a whole.

Directionality

Another important characteristics of model transformation languages is directionality. *Unidirectional* languages allow for the definition of model transformations which can be executed only in the direction for which they were thought. This is usually the case of imperative-styled languages.

Bidirectional languages allow instead the definition of transformation specifications which may be executed in both directions, i.e., from input models to output models and vice versa. This feature is extremely useful

in context in which models residing at different abstraction layers must be synchronized, for example when a designer applies some changes to the output models of a transformation and wants to push back these changes also into the input models. With a unidirectional language, two model transformation definitions are required to handle the same situation. Bidirectionality is however a feature difficult to provide and characteristic of mainly declarative-styled languages.

Cardinality

In the most simple scenario, a model transformation takes in input a single model and produces a single output model, i.e., they have a *1-to-1* cardinality. Model transformations (and languages as a consequence) may also allow for the specification of more complex scenarios. In detail, two other possible cases can be recognized: the *1-to-n* case in which from a single input model multiple output models are generated, and the *n-to-1* case in which multiple input models are used to generate a single output model. Most of the modern model transformation languages allow for the specification of all these types of transformations. This is the case for example of QVT-Operational, QVT-Relations, or ATL. Indeed, it should be noted here that the concept of separate models is rapidly becoming obsolete, as Bezivin et al. in [48] and in [49] point out by introducing the idea of *mega-model*. As a consequence, single cardinality (i.e., 1-to-1) transformation languages are not to be considered anymore less powerful than multi-cardinality languages.

2.2. Model Driven Quality Prediction

The initial purpose of MDE and of all its correlated initiatives was to provide a well understood set of principles and techniques to improve the existing practice in software and system development. However, the promotion of models to *first-class citizens* and their availability since the very first stages of the design/development process has also opened the possibility to carry out new kinds of software engineering tasks which would not be possible otherwise. An example in this sense is *quality prediction* and, in the specific case, *Model Driven Quality Prediction (MDQP)*.

Non-functional properties of software artifacts are of primary importance in many heterogeneous engineering fields. Examples range from safety-critical applications such as embedded systems controlling avionics, to wide-spread Web-based applications which we rely on everyday. End-users expect these systems to behave correctly, to be fast, and to

be reliable. Software and system developers seek for methods to deliver applications exhibiting adequate Quality of Service (QoS) which are easy to use, which bring significant development cost reductions, and which mitigate failure risks.

For example, let us consider modern Web-based systems to send and receive emails, e.g., *GMail*. We expect such systems to be fast (we tend to get upset when sending or receiving messages takes more than few seconds) and reliable (we cannot bear the loss of all our messages). On the counterpart, engineers want to be able to verify before the product is released that the system is able to handle the estimated load of concurrent requests and users, and that the probability of catastrophic failures is low. If these expectations are not met, the costs for the software engineer and for the user may be high in terms of money, time, and possibly health.

A well-known example in this sense is the *Therac-25* [50], a medical device to cure cancer with radiotherapy. The Therac-25 was involved in several accidents during the 80's which caused the death of several patients. A bug due to non-adequate testing and checking of safety properties of the device, caused the Therac-25 to expose patients to a radiation overdose of over 100-times the recommended maximum dosage, with obvious catastrophic consequences. However, this is just one of the possible examples of how non-functional properties are getting more and more important for software. The recent accident of the AirFrance Flight 447 [51] — due to both human error and to a failure in the software/hardware responsible for measuring the speed of the aircraft — is another example of a catastrophic failure which led to the death of several people.

These two examples very distant in time show an important aspect concerning software engineering and non-functional properties: if building a software artifact that provides certain functionalities is not easy, building a software artifact that exhibits certain non-functional properties is still even more complex. The core problem is that the current practice for software development tends to leave behind non-functional concerns, and to consider them only at the end of the engineering process. First the entire system is built, then its QoS is measured and, if something goes wrong, developers try to find the appropriate changes. This however leads to several drawbacks. Discovering late that a software system is not compliant with the required non-functional properties, i.e., a quality-related issue affects the software system, can be harmful for the success of the development process itself. The impact of changes on development costs and on failure risks may be indeed non negligible, if applied when a complete implementation of a system already exists.

In this sense, techniques to anticipate the assessment of quality at-

tributes in the early stages of the development process hold a lot of premise. Techniques based on very specific quality-related formalism — such as Queuing Networks (QNs) [52], Petri Nets (PNs) [53], or Markovian Models (MMs) — have been already proposed in literature. For example, QNs provide a convenient formalism to predict performance properties (e.g., response time, throughput, utilization) of enterprise systems by representing software systems as a collection of: (i) *service centers*, which model system resources, and (ii) *customers*, which represent system users or "requests" moving from one service center to another. Markovian Models such as Discrete Time Markov Chains (DTMCs) [54] and Continuous Time Markov Chains (CTMCs) have been applied to predict and verify reliability and some performance properties of software systems [55, 56, 57, 58, 59].

However these formalisms, despite their popularity in the academic community, have not yet found an appropriate placement in the current practice of software development. Only in few specialized settings — such as safety-critical systems, embedded systems, or massively distributed applications — they are consistently applied across the entire design/development cycle. The reasons of this resistance are multiple, but they can be all related to two main barriers: the difficulty of applying them and of finding the engineers with the appropriate expertise to apply them, and their perception as non-mandatory tasks which only consume time/money and do not produce significant results. Quality-related formalisms — such as QN, PN, or MM — are based on very strong theoretical and mathematical foundations, whose knowledge is necessary to understand their meaning and to correctly estimate interesting properties such as reliability or response time. The majority of the engineers *out-there* are however not familiar with all these concepts, finding the right experts is thus difficult and costs much. On the other side these formalisms are very purpose specific, they concentrate on the non-functional properties for which they were thought and they cannot be used to design all the other facets of a software system (e.g., use cases, requirements, architectures). Applications are in fact rarely represented in these terms: designers usually think at higher abstraction levels and use languages (often domain-specific) which reflect more the modeling intent such as for example UML [23] or SysML [60]. As a consequence, modeling for quality is often perceived as an extra activity which only consumes time (e.g., the time to create quality models, the time to keep them consistent with other design models).

Model Driven Quality Prediction (MDQP) techniques have born to cope with this representation mismatch and to integrate quality-prediction in a seamless manner into the design/development cycle. Germane to

many of the existing approaches proposed in literature [5, 2, 3, 61, 6, 62, 2, 3, 4, 1] is the idea to leverage MDE and MDSD techniques to raise the level of abstraction, to relieve the engineer from the burden of manually creating and maintaining consistent low-level quality models with the other system models, and to automatically derive them from higher level abstractions. Figure 2.6 depicts well this general approach. Designers work only at the abstraction levels which they are used to — possibly augmented with concepts and information about non-functional properties — and which can be used to model aspects of a system different from quality such as architectural concerns, infrastructures, functionality, or requirements. Quality models are then automatically derived from high-level abstractions, they are used as input to the various analysis tools to predict the non-functional properties of interest, and predictions are finally shown back to the engineer.

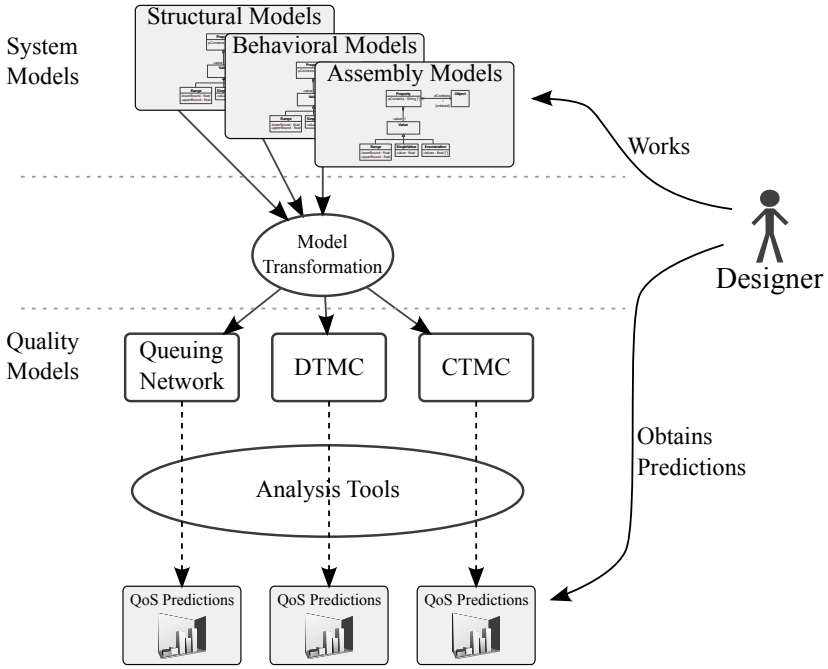

Figure 2.6.: The Model Driven Quality Prediction general framework.

A concrete example of this this approach is the Performance by Unified Modeling (PUMA) methodology described by Woodside et al. in [3]. PUMA promotes the usage of UML models augmented with information about performance by means of profiles and, in the specific case, by using the MARTE profile [63] for real-time and embedded systems. An automatic model transformation is then responsible for translating such

high-level models into a Layered Queuing Network (LQN) — a particular kind of Queuing Network specifically designed to represent and analyze multi-tier enterprise system — which may be then used to predict performance indexes of the system such as the utilization of the hardware resources, the time required to complete requests and their throughput, or the maximum number of concurrent users/requests that the designed system may sustain. Another approach similar to PUMA is the Palladio Component Model (PCM) described by Becker et al. in [2]. Instead of leveraging a general-purpose modeling language such as UML, PCM proposes its own DSL to design every aspect of a component-based systems [64], from its architecture, to its behavior, to its non-functional concerns.

Several MDQP approaches similar to these are available in literature, and all share the same *"augment with performance information and transform"* approach. Few of them however provide adequate support to understand what the numbers mean, what is the meaning of the analyses, and what to change in the system designs to overcome the potential identified issues. This problem, known as *feedback provisioning* is the problem on which we concentrate in this thesis and which we outline in the next section.

2.3. Feedback Provisioning

From the previous argumentation it may appear that models, quality concerns, and MDQP techniques play rather well together. If this was true models and MDQP techniques would have found a much broader application in industrial settings, and many of the (even severe) quality-related problems/accidents that we still experience in modern software systems would be only a matter of the past. The real situation is however much more complicated and far from this ideal situation.

Unfortunately, Figure 2.6 only depicts a part of the process. Design and development of software systems cannot be represented as a waterfall, even if we concentrate only on the non-functional aspects of a system; it is instead a cyclic and iterative process. In this sense, Figure 2.7 represents the process in a much more precise way by adding an important missing piece: a backward path from the analysis step to the modeling stage.

The vision of QoS-based and model-based development is to build a cycle where models are derived form design artifacts, and results from models are fed back to the design artifacts. Most of the activities connected to this cycle should be automated, to make it actually applicable

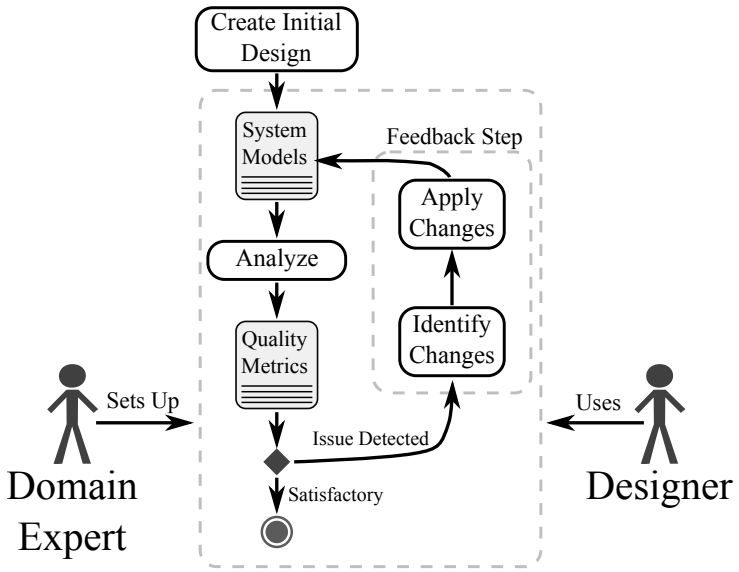

Figure 2.7.: MDQP and the feedback loop.

and able to provide timely feedbacks.

Currently, the part of the cycle from artifacts to analysis is quite well covered as we showed in the previous section while talking about MDQP techniques, whereas there is a clear lack of automation in the backward path that shall bring the analysis results back to the models. The results produced by quality analyses have to be interpreted in order to detect quality problems, if any are present. Moreover, once these problems have been detected, with a certain accuracy, somewhere in the model, solutions have to be identified and applied to remove them.

However, interpreting what the numbers produced by MDQP analyzers mean, where are the problems in the models, and which are the possible solutions is a non-trivial task. A deep understanding of the engineering domain, which the majority of developers lack, is needed, and each engineering domain is characterized by specific problems and best practices to solve issues. This is even more true if we consider that there is usually a big mismatch between the high abstraction level at which designers work and the low abstraction of the quality-prediction formalisms used by MDQP techniques and, thus, the low abstraction level at which predictions are generated and presented to designers.

The problem of looking for quality-related problems in system models and solving them is thus quite complex and is referred in literature as the *feedback provisioning* problem. Methodologies and tools to aid less experienced users in finding the most problematic areas of their designs,

in identifying solutions, and in applying them are mandatory in order to guarantee a good automation level also for the feedback step.

To clarify what *feedback provisioning* is, let us consider an hypothetical modeling setting in which a designer has to decide about the hardware infrastructure to deploy his/her Web-based system. The designer has to decide which pieces of hardware his/her company should buy, how they are connected, and how the software components should be deployed on them. All these aspects must be consciously engineered in order to meet certain quality requirements, such as the system should be responsive, the cost of the hardware should be contained, and hardware must not be over-sized. The designer is however smart and decides to use a QN-based MDQP technique to check if his/her design models exhibit the required QoS level and, unfortunately, the analysis shows that exhibited quality is not enough. For example, the results of the QN simulation may highlight that some components cannot deliver their functionalities in a reasonable amount of time for every workload situation, or that some hardware pieces are over-utilized and do not guarantee scalability when burst of requests come in. At this point, the designer finds himself/herself at the bottom of the feedback loop, he/she knows that there is a problem, but he/she has to identify it in the system designs and he/she has to identify and apply proper changes. Feedback provisioning systems come in help in this situation, by (at least partially) substituting the designer in the process of identifying and solving problems. For example, our hypothetical feedback provisioning system could propose to the designer different solutions: improving the characteristics of the problematic pieces of hardware, or changing the allocation of the problematic components on different resources, or adding new resources and balancing the requests. The more this step is automated, the more engineers can be productive, the less experience is required to produce high-quality software systems.

To sum up, this thesis concentrates exactly on the *feedback provisioning* problem, i.e., how to provide guidance to less experienced engineers when potential quality-related issues are found, multiple possible solutions are available, and these solutions must be identified and applied to system designs.

2.4. Existing Approaches

Although feedback provisioning is a relatively new area or research, different approaches have been recently proposed in literature. In this section we review and categorize them.

2.4.1. Rule-based Approaches.

A first class of solutions to automate feedback provisioning is constituted by *rule-based* approaches. As the naming suggests, rule-based approaches rely on the existence of a catalog of rules to detect in design models issues concerning non-functional properties, to identify the viable solutions, and to eventually apply them. Germane to all the approaches belonging to this category is the structure of the rules, similar to the Event Condition Action (ECA) rules [65] [66] well-known in the database and in the autonomic computing community. The feedback rules expressed according to such paradigm usually include a part responsible for detecting the occurrence of particular events in the modeling process (such as a particular design model has been produced, or a particular analysis has been carried out), a part responsible for detecting the conditions which are symptoms of a quality-related issue (such as the existence of particular patterns in the design models or of particular values in the quality predictions produced by MDQP analyzers), and a part responsible for identifying the viable solutions and of applying them to models.

The main advantage of rule-based approaches resides in their wide applicability range. As we will clarify in the next chapter, one of the most important characteristics of feedback provisioning is specificity. Every different engineering domain has its own quality metrics of interest, its own specific quality problems, and its own best practices to solve them. In this sense, rule-based approaches represent a convenient method to define all these concerns for each specific engineering domain for which feedback provisioning must be tailored. On the opposite site, rule-based approaches are also affected by non-negligible negative aspects. In detail, two are the main problems which affect them: the intrinsic difficulty of expressing the logic to identify complex solutions, and the proliferation of languages to express them.

Concerning the former problem, while writing the parts of the feedback rules responsible for identifying quality-related problems is usually simple (in many cases it is just a matter of identifying and writing the correct pattern-matching conditions), writing the part of the rules responsible of identifying the viable solutions and of applying them is much more difficult. This issue is not induced by the particular language proposed by the different approaches to define the feedback rules, but it is more related to the difficulty of formalizing, in general, what is the solution for a problem and how to apply it. In fact, in many cases, several different possibilities exist to change a design model in order to exhibit a better QoS level, these changes usually involve several parts of the design

model, require usually complex changes, and nonetheless their automatic application is not always feasible. For example, a well known source for feedback rules catalogs are performance anti-patterns [67][68][69][70], a collection of practices which should be avoided in the design/development of a software system. Anti-patterns are in many cases complicated, in many cases they do not provide a easy-to-formalize solution, and often no general solution for every problem exists.

Proliferation of languages is not a problem intrinsic of the methodology itself, but it is instead more related to how the various approaches implement it. Every solution available in literature proposes its own language to specify the feedback rules, and this poses a non-negligible barrier to their adoption. Programmers, designers, and domain experts are heavily accustomed to the tools and languages they use in their daily business. The experience necessary to master these languages to specify the feedback rules requires a non-negligible amount of time, usually incompatible with their availability of time. In this sense, we believe that rule-based approaches (as our QVT-Rational framework does) should propose the adoption of already known and used languages to express the feedback rules, in order to limit the learning curve and favor the wide-spread of such systems.

Concerning existing solutions, Xu in [8] describes a semi-automatic framework for PUMA [3] and proposes the JESS scripting language to specify feedback rules and detect/solve performance problems for enterprise systems. This approach is a *detective* approach in the sense that first problems must be introduced into the design models by an engineer, then the framework is able to detect them and propose corrections. Indeed, it has been concretely applied by formalizing some of the performance anti-patterns concerning enterprise systems available in the catalogs by Smith and Williams [67][68][69][70]. Apart from the advantages and disadvantages germane to all rule-based approaches we previously discussed, Xu's solution is affected by a problem concerning the abstraction level at which rules work. The PUMA framework leverages the LQN performance formalism to analyze some quality attributes of the system being designed. Feedback rules identify issues and propose solutions at this level; it is thus left to the engineer the task of reconciling this mismatch between the abstraction level of design models at which he/she works and the abstraction level of the information provided as feedback. In other words, it is up to the engineer to decide which modifications should be applied at the system abstraction level.

Another approach is presented by Parsons in [7], where feedback rules are derived from well-known performance anti-patterns for JavaEE-based enterprise applications. Parsons's methodology requires the existence

of a complete implementation of the software system which must be searched for quality problems, as it uses monitoring instead of MDQP techniques to estimate its performance. This is both a point in favor, as monitoring a real system usually provides more accurate estimations of the quality metrics than early model-based prediction, and a negative aspect, as it pushes the addressing of quality concerns at the very end of the development cycle. Moreover, Parsons's solution is not generic: it does not provide a language enabling the tailoring of the methodology to other engineering domains, but this solution has been thought and works only for JavaEE-based applications as the feedback rules are hard-coded.

McGregor et al. present in [71] ArchE, a programmable framework based on feedback rules to interactively and iteratively guide engineers in the generation of software architectures compliant with certain non-functional requirements. The ArchE framework encompasses several aspects of system modeling, from architecture, to requirements, to the definition of how information about quality metrics can be collected. It provides by default tools to analyze particular quality metrics (in particular performance related indexes) and suggests improvements when a bad QoS level is detected.

It is also worth citing here more recent work presented by Cortellessa et al. in [9], where a methodology is proposed to specify and identify performance anti-patterns, and to provide feedback by ranking the found issues to highlight the most reasonable causes of a performance problem. Cortellessa's approach however heavily relies on human intervention in the process (it provides thus a low automation level) and is only concerned about detection of problem. At the current stage of development, no mechanism to specify how solutions can be generated and suggested to the designer is proposed.

2.4.2. Meta-heuristic Approaches.

Meta-heuristic approaches constitute another class of methodologies to provide feedback about quality properties for system designs. This class of techniques differs from the previous one in the sense that approaches are not generic, but are tailored to the specific engineering domain for which they were thought and implemented. Meta-heuristic techniques leverage in fact particular algorithms to efficiently and quickly explore the design space in search of solutions to optimize particular quality metrics. Examples of the techniques that are usually adopted are genetic algorithms [72] and Integer Linear Programming (ILP) [73].

The main advantage of meta-heuristic approaches is efficiency. At the price of being domain-specific and in many cases sub-optimal, meta-

heuristics are rather fast in proposing solutions alternative to the initial design models which exhibit a better QoS level. As it is obvious, the main drawback of meta-heuristics is that, despite being their approach quite generic from an abstract point of view, their implementation is usually tweaked and tailored to the considered quality metrics and exploration directions (i.e., how system designs can vary or what can be changed) in order to be fast. The implementation has to leverage the available information about metrics and exploration directions in order to quickly identify alternative solutions. This severely limits the applicability of each approach belonging to this category in contexts different from the ones for which they were thought. For example, a meta-heuristic technique to optimize the performance of Web-service based systems cannot be easily adapted to other contexts such as the optimization of embedded systems.

Concerning the existing approaches, Grunske et al. in [74] survey many existing optimization approaches for real-time embedded systems. The approaches cited in this work cover several different quality metrics (e.g., from reliability, to performance, to cost and energy consumption) and different exploration dimensions. The interested reader may refer to it for further details, in the following of this section we cite only the most important and relevant approaches with respect to this thesis.

Aleti et al. in [12] present a generic framework for embedded systems, where architectural models are optimized for multiple and arbitrary quality metrics by using genetic algorithms. The framework is tailored to embedded systems and to architectural models, and at present time it considers only deployment as exploration direction. The alternative architectural models proposed by this approach may thus differ only for how components are allocated onto the hardware resources. Moreover, despite being generic, the current implementation only considers communication reliability and overhead as quality metrics.

Another similar solution is presented by Canfora et al. in [11], where an optimization framework for service-oriented architectures leveraging genetic algorithms is proposed. This framework is able to optimize a limited set of performance metrics while respecting the Service Level Agreement (SLA). The only considered exploration direction is service selection, and the performance model of candidate services is assumed to be fixed (i.e., it does not change over time and does not depend on how the service is composed with other services).

Bondarev et al in [75] describe the DeepCompass framework for design space exploration of embedded systems developed with the ROBOCOP component model [76]. This approach proposes the adoption of Pareto analysis techniques to analyze the trade-offs between conflicting proper-

ties and solutions, however no support for automatic generation of system variants is, at present time, available. The designer has to manually generate different system variants and feed the framework with them to obtain a comparison of the exhibited quality.

Kavidman et al. in [77] present another framework for optimizing Distributed Real-time Embedded (DRE) systems developed with a component-based paradigm [64]. This approach leverages model transformations to implement the heuristics responsible for optimizing an initial system design. The only considered exploration direction is again allocation, but the approach can be easily integrated with other optimization techniques (as the authors say) in order to improve its efficiency and cover other metrics/exploration directions.

More recent work is also proposed by Martens et al. in [10] where the authors describe PerOpterix, a framework to automatically improve component based systems with genetic algorithms. PerOpterix is one of the most complete solutions in this category. It supports several quality attributes (e.g., reliability, performance, costs) and several exploration dimensions (e.g., allocation, processing rates of hardware resources, replication, selection of components/hardware/middleware) to generate candidate solutions. It is tailored to the Palladio component model [2] which must be used to model every facet of the software system and, as a point in favor, metrics and exploration dimensions can be customized and extended. However, as the authors says, the framework has a questionable efficiency and optimizing an existing architecture may require an amount of time non compatible with designers' daily work.

2.4.3. Generic DSE Approaches.

Generic Design Space Exploration (DSE) approaches represent another class of systems to provide feedback about non-functional properties for generic engineering domains (i.e., they are not tailored to specific paradigms or modeling environments). DSE approaches work similarly to meta-heuristic techniques in the sense that they explore the design space in search of valid solutions (with respect to the designer's requirements) or of solutions which improve the QoS level exhibited by design models, but differ for the mechanisms proposed to explore the design space and for their *preventive* approach to quality-related issues. DSE approaches, instead of waiting for quality-issues and anti-patterns to appear into design models (i.e., for the designer to commit some errors), try to prevent their occurrence by generating from scratch system designs exhibiting the required quality. Concerning the other difference, i.e., the techniques used by generic DSE approaches to navigate the design space,

instead of using specific algorithms and heuristics with fixed exploration dimensions and quality metrics propose to encode feedback provisioning as a Constraint Satisfaction Problem (CSP), and to use reasoning techniques to solve these constraints and generate valid design alternatives.

The main disadvantage of the techniques belonging to this category resides in the languages proposed to specify how to generate system variants, or, the underlying CSP problem. The languages to specify such programs are usually logic-based; they are thus not user-friendly and require a deep understanding of the logic theory behind the encoding. This represents a important barrier for the spread of these approaches to the wide-public, which is usually acquainted with different kinds of languages and lacks the necessary background to master such formalisms.

Concerning the existing solutions, one approach which is worth mentioning here is the DESERT framework [15][78]. DESERT is not tailored to quality-driven exploration of the design space, but may be programmed in that sense. DESERT has been developed to support Software Product Lines (SPLs), and supports the exploration of design alternatives at an architectural level by organizing the system variants as a tree with boolean constraints to prune the set of viable solutions. The first proposal [15] required a massive intervention of the designer during the process; the pruning of the design space had to be performed manually by specifying which are the hard constraints to be satisfied. Its successor, DESERT-FD [78], has been designed to overcome these limitations. It comes in fact with a better and more automatic algorithm to explore the design space and provides better support for continuous finite domain design variables.

More recent work is the GDSE framework proposed by Saxena et al. in [13]. GDSE is a meta-programmable system to define and solve domain-specific DSE problems. Being meta-programmable GDSE is application domain-agnostic. GDSE provides its own language to express boolean, arithmetic, and set constraints, and supports different underlying solvers to generate candidate solutions.

Another interesting approach is the Formula framework proposed by Jackson et al. in [14]. Formula uses logic programs to specify non-functional requirements and explore the architecture design space. Models, meta-models, and constraints are represented in their Formula framework by a first-order logic with arithmetic relations. The logic program is then interpreted and solved by the Z3 Satisfiability Modulo Theory (SMT) solver, and several design alternatives compliant with the non-functional constraints are automatically generated. The system is quite fast in generating a set of possible solutions, indeed, the search of the design space is finite and to generate all the possible alternatives the

process must be iteratively repeated.

2.4.4. Quality-driven Model Transformations.

The last class of approaches for feedback provisioning is represented by those approaches using model transformations to specify the feedback rules or to navigate the design space. In the specific case, these approaches use an extension of model transformations, that is, quality-driven model transformations. Quality-driven model transformations are model transformations whose execution is directed by quality concerns, and represent a convenient mean to bind the specification of how system variants can be generated to the quality attributes of interest.

The main advantage of these techniques resides in the language proposed to define how feedback can be generated and in being generic. They in fact do not propose new ad-hoc developed languages, but they reuse existing languages usually well-known by MDE practitioners. On the other hand, the main drawback of these approaches is their performance. Model transformation engines, at least at their current stage of development, are not as fast as other methodologies (such as meta-heuristic or DSE approaches) in generating system variants. Being generic, such engines do not leverage specific knowledge about the domain or about quality (which is an advantage), hence they are not optimized and are usually outperformed by other techniques.

In detail, one of the first model transformation based methodologies is proposed by Merillinna in [16], where an approach to guide in developing architectures compliant with specified non-functional attributes is described. Merillinna's solution requires the availability of two knowledge repositories: *Stylebase*, containing the set of known architectural patterns and quality information, and *Rulebase*, a catalog of model-transformations to evolve system architectures. This approach, however, requires complete system designs and is limited only to horizontal transformations, i.e., transformation happening at the same abstraction level.

Kurtev in [17] addresses adaptability of model transformation and proposes a general framework to represent transformations with alternatives and to use quality attributes to decide among alternatives. This, however, is only a proposal; no runtime support is provided to automate the exploration process and no constructs are provided to specify how quality attributes may be computed.

More recent work is also presented by Insfrán et al. in [18], where authors propose a multi-modeling approach to select among viable alternatives (represented by different transformation rules) according to quality attributes. After the engineer has selected the desired alterna-

tives, the final transformation is derived and system models can be generated. Although in principle this approach is similar to QVT-Rational, it presents several limitations. Human intervention is required in the process, alternatives affect the solution globally (i.e., the same alternative will be selected across the whole system model), alternatives are selected a priori (i.e., before the quality attributes of the system are concretely evaluated), and it is not clear how existing model-based quality prediction techniques can be plugged in.

3. Overview of Our Approach

In this chapter we provide an in-depth discussion of the *feedback provisioning* problem and we describe from an high level of abstraction QVT-Rational, the solution we propose in this thesis to generate guidance for engineers during design and development activities. This chapter is structured as follows.

First we provide a detailed description of what feedback provisioning is and how it reifies in a MDSD setting. We delve into by describing how humans are involved in the process, by distinguishing the various roles, their responsibilities, and the time in the process at which they are involved. We pin-point important characteristics of feedback provisioning such as specificity or the various types of feedback that can be generated.

Then we provide a brief description of QVT-Rational, the solution we propose in this thesis to tackle feedback provisioning. We give an overview of the its architecture, showing what kind of models and technologies are involved in the process and how they are supposed to solve the problem of generating guidance. We also introduce some important concepts which are relevant for the comprehension of the rest of this thesis, such as the concepts of variability and variation point, and how model transformations may be used to specify how feedback can be generated.

Finally, we introduce the running example we use in the rest of this thesis to concretely show how QVT-Rational works. The example is based on the well known Object-Relational Mapping (ORM) software engineering problem, and is also one of the case studies we use in the last chapter of this thesis to demonstrate the validity of our approach.

3.1. An In-depth Discussion of Feedback Provisioning

From a conceptual point of view, feedback provisioning is all about interpretation of analysis results and generation of suggestions. Interpretation of results deals with understanding what the numbers predicted by analysis techniques mean, how they are related to the software artifacts,

what kind of problems they highlight and with which confidence they highlight them. Once potential issues are detected by analyses, generation of suggestions deals with the identification of the countermeasures to correct the identified defects or to, at least partially, mitigate them.

The general setting of feedback provisioning is depicted in Figure 3.1. As the figure suggest, feedback provisioning is a cyclic process, and this fact should be clear from this point ahead. Designing and developing modern software systems is in fact anything but easy. The complexity of hardware, middleware, and business logic, the not-easily predictable results of their interaction, as well as the complexity of the environment in which applications run make the whole design/development an iterative, complicated, and error-prone process. It is rare that a software system is correctly, with respect to both functional and non-functional requirements, designed and build at first sight; developers start from an initial solution and iteratively refine/change it until requirements are met and a satisfying solution is found. Without any kind of assistance from tools and from Integrated Development Environments (IDEs), the software engineer has to manually carry out all the steps depicted in Figure 3.1. The developer has to manually identify issues in the software artifacts, he/she has to manually find the appropriate changes to solve the identified problems, and he/she has to rely solely on its past experience to accomplish these two tasks. The more this two activities belonging to the feedback loop are automated, the more resources (i.e., money, time) can be saved, the more effective is the development process, the less experience is required on the developer side to build robust and well-designed software systems.

As a practical example, let us consider some of the code-analysis [79, 80, 81, 82] and refactoring features available in modern IDEs such as Eclipse [34] or Microsoft Visual Studio [83]. These IDEs provide easy-to-use functionalities to measure code quality and, in some cases, to identify/solve potential issues concerning code and low-level design. By estimating code quality measures [84] such as the cyclomatic complexity, the size of the method bodies, or the fan-in/fan-out of methods it is relatively easy to identify, with some degree of error, badly designed pieces of code. Some examples of problems that may be identified in this way are classes acting as *God Classes* — classes which do not respect the *Single Responsibility* principle, but which implement with a *blob* of code several interconnected functionalities used in many code locations — classes exposing the internal representation and thus violating the *Encapsulation* principle or *long method bodies*, i.e., class methods whose body is too long and complex to be easily understood and maintained over time. In addition to problem highlighting, in some cases, IDEs

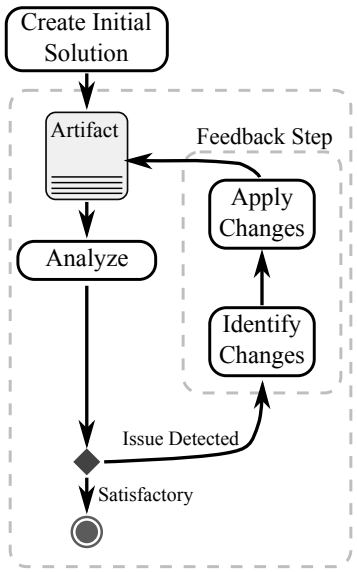

Figure 3.1.: The feedback loop.

also provide mechanisms to aid the developer in their resolution. For example, by providing IDE-aided interface extraction refactorings, the *God Class* programming anti-pattern may be partially solved; similarly, when the internal representation is exposed, IDEs provide mechanisms to automatically reduce the visibility of fields, create getters/setter (or properties in case of C#) with the necessary visibility, and change accordingly the code locations where the exposed fields were accessed.

This example, although it is not tailored to MDE or to the quality prediction techniques we described in Section 2.2, shows well the kind of situations faced daily by engineers when they need to deal with quality, it shows well some of the characteristics of the feedback provisioning process, and it gives a concrete example of the goals it tries to address. As it is obvious from this small example, feedback provisioning is anything but easy. It deals in fact with substituting developers and their expertise in very difficult tasks: problem identification and problem solving. This argument is even more true if we consider that, as we showed in the example, existing tools and techniques are not yet able to propose a solution for every kind of problem — e.g, the *long method body* anti-pattern cannot be automatically solved, only a human has the necessary understanding of which functionalities code implements and how it could be split and refactored — and that there is usually a mismatch between the abstraction level at which problems (quality metrics) are identified (estimated) and solutions proposed.

Another important characteristics of feedback provisioning shown by this small example is *specificity*. As the careful reader may have noticed, we talked about anti-patterns for Object-Oriented programming languages, and not about generic anti-patterns valid also for other programming styles such as functional or declarative styles. Also the possible solutions we mentioned are tailored to the specific Object-Oriented language considered. For example, interface extraction is a possible general solution for every Object-Oriented programming language, but using mixin-classes and traits would be another solution if supported (the Scala [85] programming language is an example of this).

All these aspects are important when dealing with feedback provisioning and in fact we considered them extensively when designing QVT-Rational. In the following, we clarify them and reify the whole approach to the MDSD setting.

3.1.1. Feedback Provisioning and MDSD

In the previous section we talked about feedback provisioning in general terms; this thesis however concentrates on a specific subset of this problem: *feedback provisioning for non-functional concerns in a Model Driven Software Development setting.*

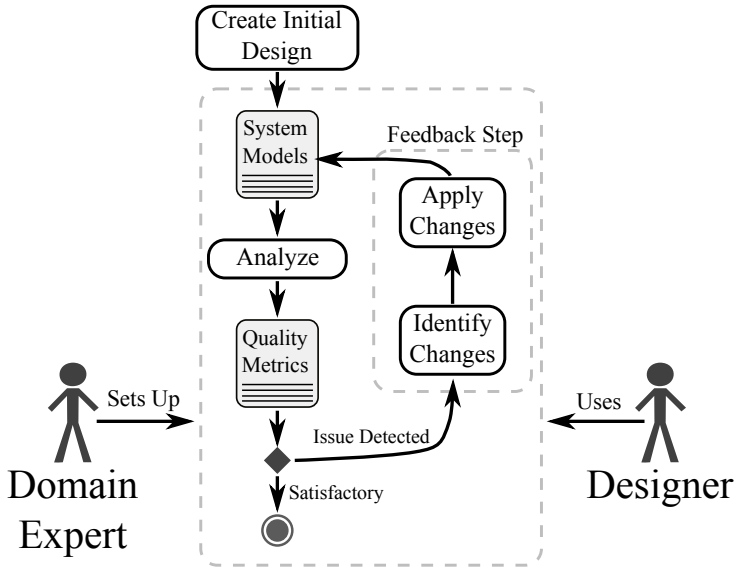

Figure 3.2.: MDQP and the feedback loop.

Figure 3.2 depicts this new setting and highlights what changes with respect to Figure 3.1. The general idea is that the system engineer de-

signs an initial version of the system, uses existing Model Driven Quality Prediction techniques to obtain estimations for the metrics of interest, and checks if the designed solution exhibits the required quality. If this is the case the process ends, otherwise the designer tries to find appropriate changes and iterates again over the loop.

This tailoring to MDSD and to MDQP introduces some important changes and raises new specific problems. In the following we discuss them.

Abstraction Level Mismatch

First of all, we must note that working in a MDSD context means substantially working with models. Everything is a model and every facet of the system should be represented in such manner, from requirements, to use-cases, to architecture, and even to code (a low level model but still a model of the application). As a consequence, we do not talk anymore about generic solutions and artifacts, but we talk about *initial designs* and about *system models*. Cycling over the feedback loop translates then into changing models, for example by refinement or abstraction, by creating new models to capture different facets of the system, either manually or using generative techniques such as model transformations to automate (at least partially) the process.

However, when working in a MDSD context, it is rare that a single model or a single kind of model (i.e., a meta-model) is sufficient to represent every aspect of the system. For example, the concepts necessary to describe requirements, the system architecture, or its behavior are quite different, and usually require different meta-models to describe them in an effective manner. Moreover, when Model Driven Quality Prediction (MDQP) techniques — such as the ones we introduced in Section 2.2 — are used to estimate the quality of system designs two other abstraction levels kick into play, the level of the quality-prediction formalism and the level of the analysis results. For example, if the engineer is interested in predicting the performance of the system being designed, the Queuing Network formalism may be used and results could be presented with the Comma Separated Values (CSV) format.

This presence of several layers of abstraction is, at the same time, an advantage for the engineer — which is provided with different languages specific for the different facets of the system that must be represented — and a disadvantage. The interpretation of the results coming from the quality analysis, i.e. checking if the Quality of Service (QoS) of the system meets the desired values is a non-trivial task, because of the abstraction mismatch among models. Each modeling notation is

intended to give a certain representation of the software system expressed in its own vocabulary. Understanding how all the models relate to each other and, in the specific case of quality prediction, how the numbers generated by the analyses relate to the modeling concepts, what they mean, and what kind of problems they highlight is not easy.

The problem of looking for quality-related problems in system models may be thus quite complex, and such search needs to be smartly driven towards the problematic areas of the model. To sum up, the complexity of this step stems from several factors: (i) quality (e.g. performance and reliability) indexes are basically numbers associated to model entities and often they have to be jointly examined to let problems emerge; (ii) quality indexes can be estimated at different levels of granularity and, as it is unrealistic to keep under control all indexes at all levels of abstraction, incomplete information often results from the model evaluation; (iii) system models can be quite complex, they involve different characteristics of a software system (such as static structure, dynamic behavior, etc.), and quality problems sometimes appear only if those characteristics are cross-checked.

Specificity

Another characteristic of MDSD is *specificity*. It has been already pointed out in literature [21, 22, 86, 87] that general purpose modeling languages such as UML [23] are not well suited in every engineering domain, while Domain Specific Languages (DSLs) are usually a better solution to guarantee the success of model-driven technologies. DSLs may be in fact created by domain experts to be as close as possible to the concepts already used by the target system designers, reducing thus the learning curve and minimizing the risk of rejection. Engineers do not want to be forced to learn and use a new paradigm to perform their daily work, they desire tools that talk in the terms they are used to and that accelerate/automate their job.

However, specificity is also a characteristic of MDQP. Every engineering domain has its own quality prediction techniques and non-functional properties which are important for system developers and end-users. For example, while developing an embedded system to control avionics an important property is reliability, and very specific formalisms to predict such property must be available (for example Markovian Models). On the contrary, if the system being developed is a massively distributed Web application (for example a social network such as Facebook), performance is much more critical than reliability; QNs may be thus more appropriate in this context with respect to Markovian Models. More-

over, given a well defined engineering domain and the set of properties of interest, the kind of development process and the meta-models used by engineers impact specificity. As we outlined in Section 2.2, every MDQP technique is thought for very specific design models. For example, if a component-based paradigm is used to design software systems, UML and the MARTE profile for the design and the PUMA approach [3] for quality prediction could be adopted. Another option would be indeed using PCM [2], its modeling concepts, and its supported quality prediction techniques.

Finally, specificity is also a characteristic of feedback provisioning. Every engineering domain is really different from each other, has its own best-practices, and has its own strategies to cope with quality issues. If we consider again massively distributed Web systems running on different platforms and middleware, the same techniques to increase reliability or performance cannot be applied in every possible situation. For example, if the underlying database storage system supports replication, this could be an option; but if it is not, other non-transparent solutions must be adopted. Another example supporting our argument is described by Parsons et al. in [88] where authors described a mechanism to detect and solve performance anti-patterns specific for JavaEE-based applications (which does not work in other different environments).

To sum up, the main consequence of specificity is that there is no general way to cope with feedback provisioning. Feedback provisioning must be tuned and programmed specifically for every engineering domain.

Human Involvement and Roles

As Figure 3.2 suggests two human parties are involved in feedback loop. The loop depicted in the figure is however different from the usual process depicted in literature about feedback provisioning. Usually the only party represented is the designer, as he/she represent the stakeholder of the entire process. In this thesis we also include in the process and highlight the role of the domain expert since, in our opinion, he/she is in charge of rather important tasks in the feedback loop.

In detail, the actors involved in the loop are the the *domain expert* and the *designer*. They are in charge of different tasks and they interact in the process at different times. The domain expert holds a relevant position in the whole process, he is in fact in charge of programming both the MDSD environment and the feedback generation strategies. We recall that each specific engineering domain has its own best practices, design meta-models, non-functional metrics of interest, and strategies to deal with quality-related issues. The domain expert possesses a deep

understanding encompassing all these aspects and we believe that he/she is the best source where knowledge about how to generate feedback can be extracted. He/she is thus in charge of sharing such knowledge by specifying how design feedback can be generated and how predictions for quality metrics can be computed.

The other human party involved in the process is the system designer, who consumes the information formalized by domain experts. The distinction between these two roles resides in the possessed knowledge; designers often do not have the same level of experience as domain experts have. Feedback provisioning systems come into play to reconcile this separation, by letting also engineers without years of development experience to adopt techniques for the resolution of non-functional issues only available to few experts.

We believe that to find an holistic solution to feedback provisioning both these roles must be taken into account. Attention cannot be paid only to the final stakeholder (i.e., the designer), but also to the domain expert in order to provide efficient and convenient tools and mechanisms to define how feedback can be generated.

Types of Feedback

The last aspect that needs to the tackled about feedback provisioning is the *what*, or what concretely feedback is. What kind of information is actually conveyed by feedback tightly depends on the concrete methodologies proposed to generate feedback. For example, meta-heuristic approaches generally work by improving existing system designs as specified by the adopted heuristic and propose feedback in the form of complete system variants optimized for some quality properties. On the counterpart, rule-based approaches usually concentrate instead on problem identification and, if possible, changes to cope with the found problems will be consequently proposed. Feedback consists thus in the list of problems found in system designs and, possibly, in the list of resolution actions that may be performed by the designer.

In this sense, the different types of feedback that can be proposed to designers can be organized along three dimensions, as depicted in Figure 3.3. On the first axis, feedback can be classified according to how feedback is concretely presented, or the *medium* used to convey it. Depending on the kind of problems a feedback provisioning system is able to address, on the intended audience, and on the specific engineering domain it is tailored to, different ways to convey feedback information are viable and range from natural language to formal models. For example, feedback may be produced in the form of a general description of the

problem and of the possible resolution actions. This is especially true when there is no general way to identify a solution for every specific engineering domain, but intellectual work is required on the designer side to find it. On the other side, feedback may be also conveyed in a formal way by using models. For example, new system models exhibiting the changes appropriate to solve the identified problems may be proposed as feedback.

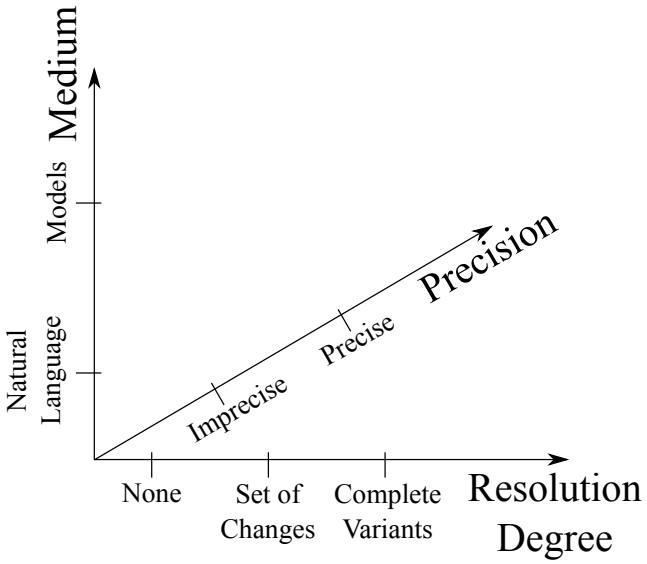

Figure 3.3.: Feedback types dimensions.

The second dimension is *precision*, a dimension tightly tailored to the previous one and which deals with how precise is the information conveyed, irrespective of the medium (e.g., natural language or formal models) concretely used to covey it. In fact, natural language does not implicitly mean that the information contained in a piece of text is not formal. In the same way, using models does not implicitly entails that for every possible issue a complete solution can be identified and presented.

The third and last dimension is the *resolution degree*, which concerns how much information about the possible solutions to the identified quality-related issues is conveyed by feedback. The resolution degree may spawn from the lower bound of no information at all — when quality-related issues may be only identified but no specific suggestions (apart from general information) can be proposed — to the case in which the list of changes to apply to system models are presented, to the upper bound which consists in the case when the, possibly multiple, viable system alternatives are created and proposed.

As a general though, the more precise, formal, and complete (with respect to the resolution degree dimension) feedback is proposed the more effective is a methodology. These goals are however not as simple to reach as they may appear. Nonetheless, the placement of most of the approaches we described in Section 2.4 is in the middle. Those which excel in some dimension reach that level at the expense of the other. For example, meta-heuristic approaches which excel in the *medium* and *resolution degree* are usually imprecise in the sense that the found solutions are local optimums.

3.2. Introducing QVT-Rational

QVT-Rational is our proposal for a feedback provisioning system. As we clarified in the previous sections, when feedback provisioning is considered in a Model Driven Software Development setting several are the important characteristics that must be taken into account for the success of the approach. For example, we mentioned how feedback is specific in the sense that it must be tailored to every engineering domain for which it must be generated, we mentioned that multiple human parties with different roles and interaction times are usually involved in the process, and we clarified that several different types of feedback can be generated.

While designing our solution, we took into account all these aspects and the outcoming architecture of QVT-Rational is well depicted by Figure 3.4. QVT-Rational is a **multi-modeling**, **multi-target**, **programmable** framework to specify how to generate and to produce feedback for system designers in the form of complete system variants satisfying certain non-functional requirements.

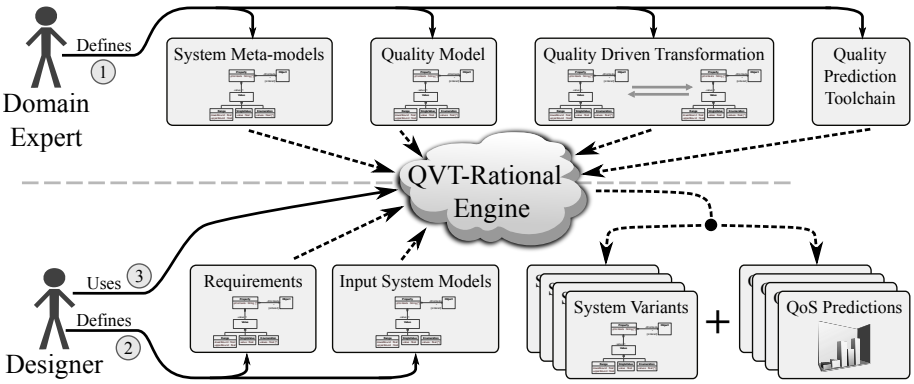

Figure 3.4.: Overview of QVT-Rational.

Multi-modeling means that different models and languages concur in

the feedback provisioning-enabled MDSD environment, programmable means instead that our system can be tailored to the needs of every different engineering domain. The direct consequence of specificity is that each engineering domain has different modeling abstractions, different quality properties which may be of interest, different methodologies to predict them, different requirements, and finally different quality-related issues and strategies to cope with them. All these aspects can be configured in our solution and, being QVT-Rational a Model Driven Engineering methodology, we use models to express such information in a formal and precise manner.

Multi-target means instead that different parties are involved in the feedback provisioning process. As we outlined in the previous sections we identify two fundamental roles (the *Domain Expert* and the *Designer*), and this distinction is reflected in the architecture of QVT-Rational. The domain expert (depicted on the upper part of Figure 3.4) is in charge of specifying the modeling abstractions characteristic of the engineering domain, of specifying the quality properties which may be of interest, how to evaluate them via MDQP tool-chains, and how feedback can be generated (which, as we will see in a short, can be accomplished by using extended model transformations). The designer (depicted on the lower part of Figure 3.4) interacts with QVT-Rational by asking for feedback: by specifying which are the existing software system models for which he wishes to obtain suggestions, by determining the required level of quality, and by obtaining from our system feedback in the form of the viable design variants which exhibit the required quality.

Besides providing an integrated environment encompassing all the characteristics specific of feedback provisioning for quality concerns, one of the major novelties of QVT-Rational is the mechanism we propose to specify how feedback can be generated. Our approach is hybrid with respect to the classification proposed in Section 2.4. It is between the realm of rule-based approaches and the realm of model transformations based approaches. We in fact propose our quality-driven model transformation language to specify the feedback rules. In the following we concentrate on this aspect.

3.2.1. Quality-Driven Model Transformations for Feedback

One of the major drawbacks of existing rule-based techniques resides in the fact that every approach proposes its own language to specify the feedback rules. For example, Xu in [8] uses JESS to write the rules responsible for matching problematic patterns into design models and of identifying the possible reconfiguration actions. Another example is

the work proposed by Cortellessa et al. in [9] which uses an XML based language to specify the feedback rules.

The shortcoming of this situation is that to use such approaches new languages must be learned and mastered, which is not something advisable in industrial settings where productivity is a major concern. Nonetheless we recap that as Hettel points out in [39] model transformations encapsulate the rationale behind the design process, they also promote knowledge reuse and sharing, and they are the preferred mechanism to improve productivity in an MDSD setting.

As a consequence of this, our idea is to use exactly model transformations to specify the feedback rules. Many feedback provisioning systems produce system variants (i.e., models) as the output, also QVT-Rational follows this pattern, and model transformations produce models.

However, existing model transformation transformation languages cannot be used as-is to specify the rules to generate feedback. First of all, existing languages lack the appropriate concepts to include quality concerns in the transformation source code. As a matter of fact, what we propose is the usage of the so-called *quality-driven model transformations* [16, 17, 18, 19, 20], i.e., transformations which embed information about quality concerns for the input/output models and whose execution is driven by that information.

Second, we mentioned before that QVT-Rational produces feedback in the form of complete system variants exhibiting possibly different quality. To express the feedback rules (or the rules to generate these design alternatives) we use non-injective model transformations (transformations which may produce different outputs for the same inputs). However, the existing transformation languages (and their associated execution semantics) do not deal well with non-injection. In many cases (for example for imperative-styled languages) transformations with alternatives cannot be easily expressed (or expressed at all). For declarative-styled languages the execution semantics is usually the problem: the first solution found is a valid solution. We need instead to control how alternatives are picked-up, discern them according to their quality, and potentially produce any viable output.

To achieve such goals, QVT-Rational proposes an extension of existing model transformation languages with the appropriate construct to elicit quality concerns and alternatives, and proposes its own execution semantics to navigate the space of the possible transformation alternatives.

Variability and Variation Points

As we just outlined, we specify feedback through non-injective model transformations able to generate multiple design alternatives with different impact on the non-functional properties of the system. We achieve this goal by extending existing model transformation languages with appropriate constructs and runtime support. The concepts of alternatives, when/where they occur, and of deciding among them play a central role both in the problem statement and, as we will see, in the solution we propose. Most of these concepts, however, are not new, but have been already addressed in the literature about SPLs [89].

In terms of SPLs, what we informally called alternatives correspond to *variant*s, while the point at which they occur are called *variation point*s. Many definitions have been given in the past for these terms. For variation points, we stick to the definition given by Jacobson in [90]: *"A variation point identifies one or more locations at which the variation will occur"*. A variation point defines the *where*, or the context in which the variability happens. A variant instead represents *how* software artifacts vary, by specifying which are the possible alternatives.

Variability and MDSD

When variability issues are tailored to MDSD, different kinds of variability may be recognized. We stick to the classification proposed by Sijtema in [91], which identifies two categories: source model independent variability and source model dependent variability. Variability is source model dependent if the selection between variants depends on information contained into source models, such as the existence of particular objects or certain annotations. On the contrary, if this condition does not hold the variability is said to be model independent. An example may help to understand this classification. Let us consider the problem of selecting among different database technologies, such as a relational database or an XML-based database. The selection of which technology to adopt for a system is orthogonal and unique with respect to the application model. Either we select relational or we select XML-based for the entire application, hence this variability is model independent. On the contrary, let us consider the problem of deploying a Web application on an infrastructure comprising many servers. How each software component is assigned onto each server may have an impact on the performance of the whole application, depending on how the components interact. The decision about the variability depends thus on the context in which it is taken, and belongs to the model dependent category.

3.2.2. Goals

To sum up, QVT-Rational addresses the feedback provisioning problem from the perspective of all the parties involved in the process. It provides a convenient model to define quality properties, a language to define non-functional requirements, and provides an engine to automatically explore solutions and provide guidance to engineers. The advantages of QVT-Rational with respect to the existing approaches are several and the goals it tries to achieve are:

- **Language Uniformity:** by using widely-adopted model transformation languages to define solutions to quality issues, domain experts may reuse their knowledge and do not need to master new approaches and specific languages.
- **Quality Metrics Support:** our approach is not limited to specific metrics.
- **Environments and Tool-chains Reuse:** existing modeling environment and quality prediction tools may be plugged in as-is into our framework.
- **Automation:** the availability of an automatic, requirements-driven, exploration engine can bypass the engineer in the feedback loop.
- **High Abstraction Level:** feedback is generated and presented at the abstraction level at which the engineer works. End users are not required to know all the details of the underlying quality prediction methodologies.

3.3. Introducing the Running Example

In this section we describe the running example we use in the rest of this thesis to show how QVT-Rational helps domain experts specify the feedback rules and how quality-driven model transformations are actually executed in order to generate suggestions for system designers. We anticipate here that the example described in this section is also used in the last chapter of this thesis to demonstrate the validity of our approach.

The running example we propose tackles the Object-Relational Mapping (ORM) problem. Many software systems use RDBMS technologies to store business relevant information, but architects and developers talk in terms of domain entities. Nonetheless, also source code talks in terms of domain entities, especially when modern middleware is used to handle persistence. In support of this argument, we may consider the JavaEE

[26] framework and the Enterprise Java Beans (EJB) standard. Domain classes are represented as plain classes in the source code, annotated with special constructs to indicate that they represent the domain entities and that they should be managed by the framework accordingly.

However, even if this convergence between the design level, the source code level, and the database layer represents an important improvement in terms of code maintainability, readability, and understandability, some aspects of the mapping between these layers cannot be totally handled in an automatic way by the underlying middleware. The two representation levels are in fact not compatible in every aspect; ORM deals with this mismatch between the relational level and application domain models, by providing a methodology to translate between the two representations and correlate their respective concepts.

Adopting Object-Relational Mapping as the basis for our example brings several advantages. First, ORM is a software engineering problem, widely-known both in academia and in industry. The majority of practitioners and researchers know in detail the problem, its caveats, its solutions, and this favors comprehension. Many examples taken from real-world applications are also publicly available, such as [92][93]. It is indeed easy to find use cases with the complexity necessary to show if our approach can handle non-trivial situations.

The selection of this problem also favors comparisons with other approaches. Mapping entity models to relational models is a common use case in literature about model transformations. Examples are the publications about ATL [94], TGGs [95] and the QVT specification [41]. ORM has been adopted also to show quality-driven model transformation approaches [18].

Finally, ORM also fits well our variability and quality prediction needs. A hypothetical example for feedback provisioning must show points of variability, and methodologies to evaluate the quality of the various alternatives should be available. Indeed, performance evaluation of databases has been widely studied in literature. A survey has been conducted by Nicola and Jarke in [96] and existing approaches can be adopted in our case to generate feedback. Concerning variabilities, the standard methodology to cope with ORM presents many of them. Two examples are inheritance mapping and generation of row identifiers. Hierarchies of classes may be mapped according to three different strategies — table per class hierarchy, table per concrete class, and the joined subclass — all with different characteristics and impact on the performance of the database. The same holds for identifiers, which may be generated in a sequential manner, by pre-allocating them, or by using more complex algorithms.

In the rest of this section, we delve into the various aspects that we just mentioned. First we show how domain entities and databases may be modeled, i.e., we show the meta-models we use in our example. Then we concentrate on the quality aspect we are interested in and how they can be evaluated. Finally, we provide a brief description of the variabilities we consider in the example and we describe how they impact quality metrics.

3.3.1. Modeling Domain Entities and Databases

Being QVT-Rational a model-driven approach, the first aspect of the ORM example that must be clarified is how domain entities and databases may be represented with models. In particular, we need to define the meta-models that we use for this purpose and the concepts they provide.

As we mentioned before, ORM is a widely adopted example in the literature about model transformation. It is then no surprise that it is also one of the examples used in the QVT [41] model transformation language specification proposed by the OMG. The ORM example we propose is based on the one presented in such specification, tailored to our needs and extended in order to be able to include variabilities and quality concerns.

Figure 3.5 shows the meta-model we adopt in the example to design domain entities. The natural way to specify domain entities for a particular application is to use a UML class model. This is also the solution proposed in the original version of the ORM example available in the QVT specification. Note however that, since we are interested only in modeling domain entities, we do not use a complete class meta-model, but we concentrate only on the concepts and on the aspects relevant for this purpose. As a matter of fact, in this reduced class meta-model named *Simple-UML class* domain models are specified as a *Package* containing *Class*es and *PrimitiveDataType*s (for example *String*s, *Integer*s, or *Boolean*s). *Class*es may be connected through *Association*s and through *Generalization*s in order to specify a type-hierarchy between domain entities.

In order to tailor the meta-model to our ORM needs, we added two modifications to the original meta-model concerning the cardinality of *Association*s and the specification of identifier properties. For what concerns cardinality, we added two attributes (i.e., *sourceCardinality* and *targetCardinality*) which specify, respectively, the cardinality with which classes participate to *Association*s. For what concerns identifiers, the *isId* flag attribute has been added to the *Attribute* meta-class to specify when a property acts as an identifier for a particular domain entity.

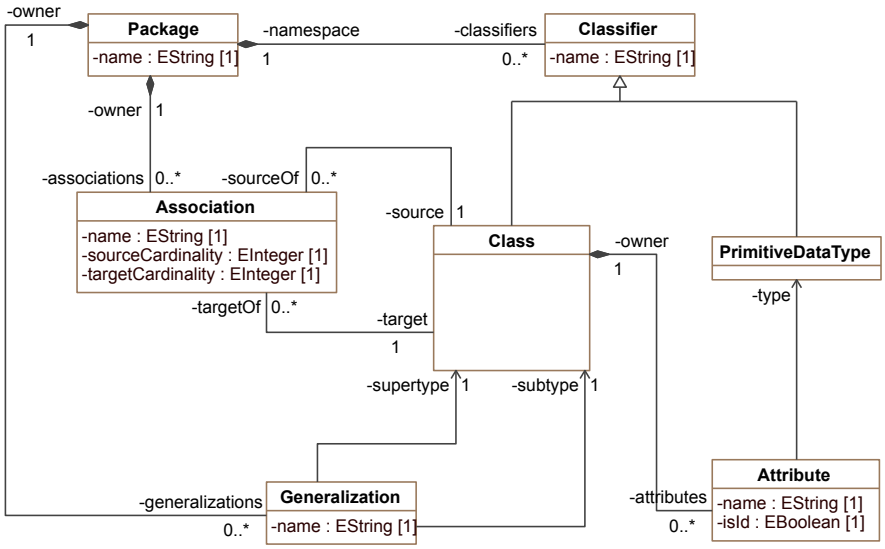

Figure 3.5.: The *Simple-UML Class* meta-model.

For what concerns database modeling, the meta-model we adopt is depicted in Figure 3.6. A database is represented as a set of *Table*s belonging to a *Schema*. *Table*s may contain *Column*s of various types (for example *CHAR*, *INTEGER*, or *DATETIME*), and *Column*s may belong to *Key*s and to *Foreing Key*s. In the former case, they define the record identifiers for a particular table, in the latter they specify how tables are connected by relationships. In addition to these concepts that are also defined in the original example available in the QVT specification, we added to the RDBMS meta-model the concept of *Partition* and we added to the *Column* meta-class the *autoGenerate* attribute. These two modification have been applied, as we will clarify shortly, in order to be able to represent two well known techniques to cope with database performance issues, i.e., horizontal partitioning and scattering of record identifiers.

As the name suggests, the *Partition* concept serves the purpose of representing databases in which the data of some entities is split across several tables according to some partitioning policy (for example by dividing the primary keys into disjoint ranges and assigning to each table in the partition a different range). *Partition*s may thus contain several tables, which we distinguish by assigning different values for the *sliceId* attribute. In order to clarify this aspect, let us consider a hypothetical database for an e-commerce application, and let us suppose that the

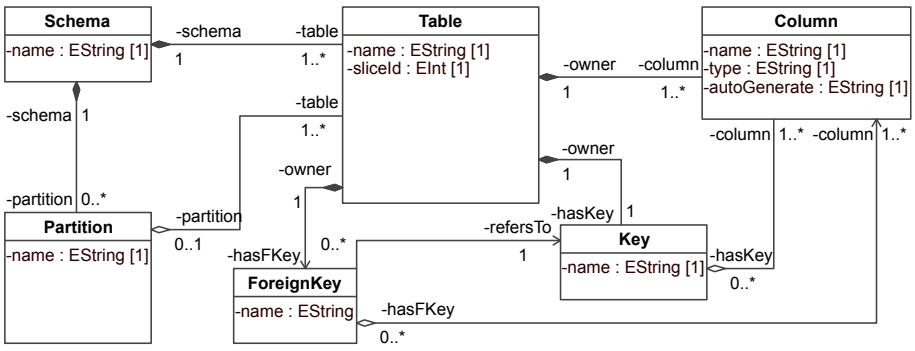

Figure 3.6.: The extended *RDBMS* meta-model.

Product domain entity, given the huge number of records it may contain, must be represented with a partition of tables in order to speed-up queries. Figure 3.7 shows how this would be represented with our extended RDBMS meta-model. A partition named *Product* will represent the domain entity and will contain several tables all equal apart from the assigned *sliceId*.

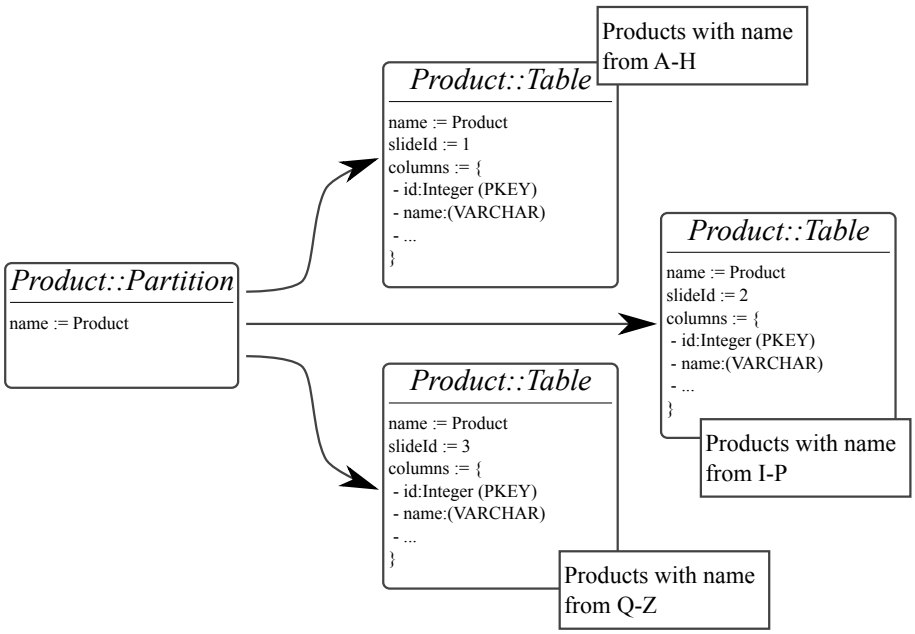

Figure 3.7.: An example of database with partitioning.

The *autoGenerate* attribute for the *Column* meta-class serves instead the purpose of specifying which policy should be used by the database manager to auto-generate values for columns participating to record iden-

tifiers (i.e., primary keys). For reasons which will be clear shortly, using strategies different from the standard sequential one (i.e., values are generated sequentially) brings performance advantages under special circumstances.

3.3.2. Quality Properties and Prediction

Since we deal with feedback for quality concerns, the other aspects which we should clarify are what metrics we consider in the example to evaluate the quality of a database schema and how we may predict them using MDQP techniques. This is important in order to show how QVT-Rational takes care also of the quality prediction step in the feedback loop.

For the sake of clarity, in our running example we concentrate on two quality metrics for database schemas: the time required to perform read/write operations (i.e., the *query response time*) and the *wasted-space*.

Wasted-space deals with space allocated for records by a database system which is not actually used. Some design decisions, in order to favor performance of critical queries, are affected by this problem. For example, as we will see next while talking about inheritance, the *upward-collapse* strategy favors performance by decreasing the time required to read records — joins between tables are not required to retrieve all the entity columns — but the space occupied by each record sums up to the space required to store all the properties of every class involved in the hierarchy, even if a specific object instance does not have all of them. To predict the wasted space of a table we compute the ratio between the space used by each record to actually store the properties of a class (discounted by the frequency with which instances of the class occur with respect to the other classes mapped onto the same table) and the space allocated by the database system for the record. Equation 3.1 shows the exact formula we use to estimate wasted-space for a given table. The $freq(c)$ function retrieves the frequency with which instances of a given class occur with respect to other classes in a hierarchy. If a class does not participate in an hierarchy, its frequency is indeed one. The $map(t)$ function computes the set of classes mapped onto table t, $cols(c, t)$ computes the set of columns used in table t to map the properties of class c, and $cols(t)$ retrieves the set of columns defined in table t. The space required to store a column cl is instead computed by the $space(cl)$ function. Wasted-space ws can be then computed as the discounted ratio between the space used by each record to actually store the properties of a class and the space allocated by the database system

for the record. The lower the values, the less space is wasted, the more desirable is the database design.

$$ws(t) = \sum_{c \in map(t)} freq(c) * [1 - \frac{\sum_{cl \in cols(c,t)} space(cl)}{\sum_{cl \in cols(t)} space(cl)}] \tag{3.1}$$

The other non-functional property of interest is the time required to execute read/write operations, or the query response time. Several approaches have been proposed in literature to perform this kind of estimation for databases [96]. Among these approaches, the ones based on Queuing Networks fit well our needs: the final database implementation is not required but schema models are sufficient for the prediction, the kind of queries performed on database are taken into consideration for the analysis, and tools to simulate and analyze QNs are publicly available [97]. In detail, the methodology we adopt is based on the approach presented in [98]. As shown by Equation 3.2, the response time of a query, performing either read or write operations on several tables, may be expressed as the sum of three different factors: the time to wait for accessing tables, the time to access the data on tables, and the time to return data to the client.

$$\begin{aligned} rt(q) = \sum_{t \in tables(q)} [&wait\_time(t) + \\ &access\_time(t) + \\ &return\_time(t)] \end{aligned} \tag{3.2}$$

All such concepts map straightforward to QNs: tables may be represented as service centers with First Come First Served (FCFS) discipline, and, in case of partitioning, as a set of service centers. The wait time for accessing a table corresponds to the queuing time, the data access time corresponds to the service time, while the return time can be represented with a delay center. Different queries are mapped to different customer classes, whose routing through the network depends on the tables accessed by the queries themselves. The response time of a query can be indeed measured as the total time required by requests to traverse the whole network. For each query (i.e., customer class) three parameters must be also determined: the workload type, its characteristics — arrival-rate in case of open workloads, think-time and population in case of closed workloads — and service demands for each table. All of them depend on information provided by engineers as part of the usage profile for the database. The workload type is specified as-is by designers, service demands are instead approximated with an exponential distribution, whose mean is equal to the number of memory pages accessed

by the query for that particular table [96]. For simplicity, we assume that this parameter is provided as-is in the usage profile; techniques to estimate it from the query structure are indeed available [96].

3.3.3. Variabilities and Impact on Quality

Given an entity model specified via the *Simple-UML class model* ORM deals with translating it into a corresponding database model. When MDSD generative techniques such as model transformation are used, this translation can be automated at least partially. The idea is that the transformation logic can be specified in a model transformation so that the developer may be relieved from the burden of manually performing the transformation.

The transformation we use in our example is, as it is clear now, taken from the QVT specification. This transformation is a reduced version of the general ORM translation problem. An ORM translation could be in fact quite complex if we consider all the possible strategies to perform it and all the possible optimizations. Nonetheless, even by considering only the most straightforward translation strategies — mapping *Class*es onto *Table*s, *Attribute*s onto *Column*s, *Association*s onto *ForeignKey*s, and generalization with the upward-collapse and downward-collapse strategies — the model transformation to automate the process is non-trivial.

As we mentioned in the previous Sections, in order to generate feedback we require the availability of a non-injective transformation, that is, a transformation able to produce different outputs given the same inputs. Since the original ORM transformation proposed in the QVT specification does not exhibit such characteristic, we extended it with the logics to represent three variabilities which impact the non-functional properties of the produced database schema. In the rest of this section, we describe them in detail.

Inheritance Mapping

The first variability we introduced concerns how inheritance hierarchies are mapped. In literature, three are the strategies available to cope with them: table per class hierarchy (upward-collapse), table per concrete class (downward-collapse), and joined subclass.

The downward-collapse solution maps each entity participating in the hierarchy to a separate database table with columns to represent every property, including inherited ones. This strategy poses polymorphism and performance problems [92] and is not suggested by ORM guidelines, hence we deliberately excluded it from our example.

The upward-collapse strategy suggests the usage of a single table for each inheritance hierarchy, all the properties of each entity participating in the hierarchy are mapped onto the table and a particular column, the discriminator, is used to distinguish among instance types. This strategy performs well in situations where read operations outnumber write operations, but it is characterized by space waste. The size of each record is *fixed* and space for each column is reserved even if it is not used by a specific record.

The joined-subclass strategy maps each class type on a different table containing columns only for the type specific properties, and inheritance is represented by using relationships between parent and child types. Conversely, this solution does not suffer from space waste problems, performs well when write operations are more frequent, but read operations require searching and joining across different tables.

Identifiers Generation

This variability has been inspired by a particular performance anti-pattern, the *one-lane-bridge* [68, 69, 70, 67], and concerns how record identifiers are automatically generated by the database system. The one-lane-bridge occurs in a software system when "*at a point in the execution only one, or a few, processes may continue to execute concurrently*". For database systems, this happens when multiple clients concurrently access and insert/modify records in the same table, that is, they access the same domain entity. Different solutions have been proposed to mitigate the problem, and changing the default policy used by RDBMSs to generate record identifiers is one of them. When the default sequential algorithm is used for this purpose, sequentially inserted records have close identifiers and share the same storage location (or memory page). When inserting a new record, the database engine needs to lock the entire page to store the new entry, indeed blocking other clients. By using different strategies, such as *randomized* generation, clients may be diverted to different storage locations and may consequently proceed concurrently.

Partitioning

Another solution to the one-lane-bridge anti-pattern leverages data partitioning. If data is split across different tables, multiple clients may proceed in parallel as long as the records they are accessing reside on different slices, and access time can be consequently reduced. The underlying idea is then, instead of translating a domain entity onto a single database table, to translate a domain entity onto a partition of tables as

we described in Section 3.3.1. By doing so, requests accessing different data partitions may proceed in parallel with obvious nice side-effects on the performance of the database.

4. Programming The Framework

In this chapter we describe how QVT-Rational helps domain experts in the specification of the feedback rules. Here we concentrate only on this aspect, how quality-driven model transformations are actually executed in order to generate suggestions is a matter for the the next chapter. This chapter is structured as follows.

First we give a brief recap on the role of the domain expert and the tasks he is responsible for when using QVT-Rational. We first delve into the core of QVT-Rational, that is, how model transformations are extended with appropriate constructs to elicit variability and feedback rules. We introduce our simple annotation language, the two model transformation languages we extend, and we show how they can be concretely used to specify feedback rules.

Then we then describe how non-functional properties may be specified, that is, the meta-model we propose and the available concepts, and we show how variabilities expressed in model transformations can be bound to quality properties. Finally, we demonstrate how we support the definition of complex constraints between design variabilities, a mandatory feature in order to be able to manage also non-trivial modeling scenarios.

All the argumentation in this chapter is reified by means of the ORM example we described in the previous chapter.

4.1. Recap on the Domain Expert Role

In the previous chapter we introduced QVT-Rational, we described the idea of using quality-driven model transformations to specify the feedback rules, and we outlined the various steps, artifacts, and human roles involved in the process. We used Figure 4.1, which we report here for comprehension, to show how all the pieces constituting the solution are interconnected.

In this chapter we concentrate on the upper part of the figure, that is, all the artifacts that are produced by the domain expert. We recap that the idea of QVT-Rational is to partially shift the responsibility of finding solutions to quality-related issues (i.e., problems concerning performance, reliability, or non-met requirements about non-functional properties) from the system designer — who may not be aware of all the

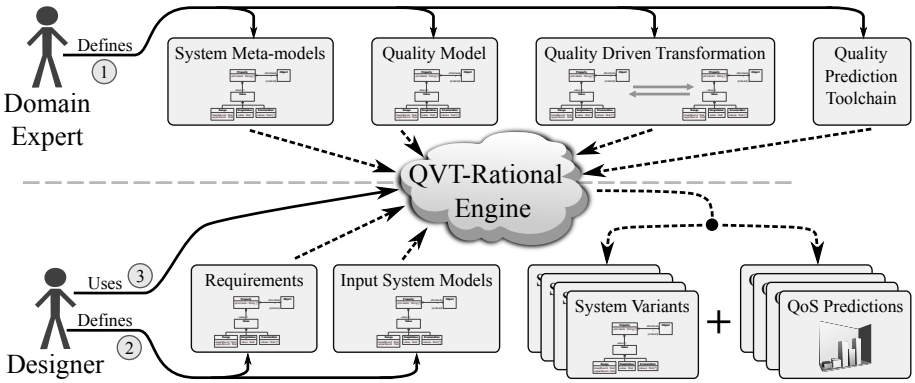

Figure 4.1.: Overview of QVT-Rational.

techniques and solutions available to cope with non-functional concerns — to the domain expert — the best source for this kind of information about techniques and solutions. In detail, the various tasks and artifacts that must be produced by the domain expert before QVT-Rational can be concretely used by engineers to get feedback are:

- **Defining *System Meta-Models***. Our approach is a Model-Driven approach and, as a consequence, designing and developing systems means working with models. However, every engineering domain has its own practices, formalisms, and concepts as we mentioned in Section 3.1.1 while talking about specificity. As a consequence, one of the duties of the domain expert is to specify the meta-models and the DSLs that will be used by system designers. However, designing the DSLs is not the only task required to *program* and prepare an MDSD environment. For example, rich editors for editing models must be developed, model transformations to automate some of the development steps must be created, and in general a lot of customized tools must be created in order to assist the end-users of MDSD environment. For example, if we consider our ORM running example, preparing the MDSD environment would require designing the *Simple-UML Class* and the *RDBMS* meta-models, the editors to create instances of those models, the tools to check their validity, the transformations to automate the translation between them or to derive the SQL commands to concretely build database schemas. Even if tools that automate and provide support are already available on the market — some notable examples are EMF[33], the VSVMSDK[35] or MetaEdit+[99] — these tasks are obviously non-trivial and require a lot of work. They are however out of scope for this thesis,

where we concentrate on the feedback provisioning problem. As a consequence, we do not dig into the details, but we safely consider that a standard MDSD development environment (without feedback provisioning capabilities) has been already developed for us.

- **Defining the *Quality Model***. We specifically consider the feedback provisioning problem in the context of non-functional concerns. Quality is thus a central aspect in our approach and needs a formal definition. With formal, we mean that the various quality properties which might be of interest for a particular engineering domain must be specified, that they must be correlated to the domain concepts provided by system meta-models, and that they must be bound to the feedback rules specified in the quality-driven model transformation. The domain expert is in charge of specifying all such information, as he/she is the best source of knowledge about it. We dig into the details of this topic in Section 4.4, where we describe the meta-model provided by QVT-Rational to define quality metrics and how properties can be bound to the feedback rules.

- **Defining the *Quality-Driven Model Transformation***. The core idea behind our approach is to define the rules to generate feedback by means of quality-driven model transformations. As we will describe in detail in Section 4.3, to generate feedback we rely on the existence of a non-injective model transformation where variabilities are annotated with appropriate constructs to elicit them and to define their impact on quality. The domain expert is responsible for creating such a transformation, and QVT-Rational provides an extension of two existing model transformation languages for such purpose. In the following we introduce these languages, we show how non-injective transformations with variabilities can be specified, and we describe the annotations extending the original model transformation languages.

- **Defining the *Quality Prediction Tool-chain***. Dealing with quality (especially in the early stages of the development process) means that mechanisms to estimate it from the available software artifacts must be available. In our case, we work in an MDSD setting, our artifacts will be models, and MDQP techniques can be used for the purpose of obtaining quality predictions. As it is obvious from Figure 3.4, the domain expert is in charge of identifying which methodology should be adopted to estimate the vari-

ous quality properties of interest and of building the appropriate tools to concretely compute predictions from system models. A consequence of specificity is that every engineering domain has its own techniques and formalisms, hence, specifying and creating the MDQP tool-chain can be thought as a part of the MDE environment programming. As we did for the specification of the system meta-models step, we do not dig into the details of this step, but we just act as end-users. We suppose that MDQP tool-chains already exist (or are created by the domain expert) and we reuse them into our approach to generate feedback about quality.

4.2. Introducing the QVT Family of Transformation Languages

It should be clear now to the reader that a key artifact in QVT-Rational is the quality-driven model transformation specifying how feedback is generated, or the rules to generate different system variants with different non-functional properties. As we mentioned in Section 3.2, we believe that model transformations are the best medium to convey such information. Model transformations are *"the heart and soul of Model Driven Software Development"* [37], they can be considered as *"definitions of the engineering process"* [39], and using them to specify how to generate system alternatives when non-functional requirements are not met brings several advantages. One of them which particularly impacts the domain expert is that, instead of using ad-hoc languages to specify the feedback rules as many other approaches do, languages well known in the MDE community — and thus by domain experts already working in a MDSD setting — are used; no new languages need to be mastered with consequent reduction of the learning curve.

In the specific case of QVT-Rational, to define feedback we extend two well known model transformation languages: QVT-Operational (QVT-O) and QVT-Relations (QVT-R). Both these languages belong to the QVT family of model transformation languages [41] proposed by the OMG. We opted for this two languages and not for other proposals [46, 44, 43, 45, 42, 95] only for convenience. Being an OMG proposal, the chances of a wide adoption of such languages and are higher than for the other languages. Another consequence is that the chances of being maintained for a long period are higher. Moreover, supported tools and editors exist for the Eclipse platform.

Figure 4.2 provides an overview of the various constituents of the OMG proposal. Altogether QVT is composed by three languages: two of them

(QVT-O and QVT-R) are thought for direct usage by engineers, the remaining one (QVT-Core) provides instead a common ground to define the execution semantics of Relational transformations.

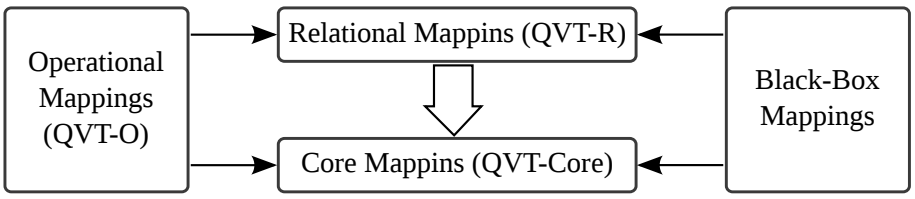

Figure 4.2.: The QVT family of model transformation languages.

QVT-Operational is an imperative-styled language to define mappings, i.e., how input/output model elements match to each other. Given its syntax and semantics closer to widely-adopted programming languages such as Java or C#, QVT-Operational is also the most adopted language proposed in the QVT specification and, as a matter of fact, is also the most supported one (an efficient engine, editors, debuggers, and tools exist for the Eclipse platform [34]).

QVT-Relations proposes instead a declarative-styled language to define mappings. It is based on a complex pattern matching mechanism and, as we will see in a short, transformations based on it may be thought as traditional textual grammars [100]. Given its style different from the one usually adopted for programming languages and the difficulty of specifying complex transformations (with respect to its operational counterpart), its adoption is still in the early stages and few tools and execution engines exist [101, 102].

The last boxes in the figures which need an explanation are the box representing the QVT-Core language and black-box mappings. As we mentioned before, QVT-Core serves the purpose of defining a common ground to specify the execution semantics of QVT-Relations. Its structure is rather similar to the one proposed for QVT-Relations, in fact it is equally powerful to the Relations language, but it has a simpler semantics, it has a simpler syntax, and it is more verbose. QVT-Core is not intended for direct usage by humans; its main purpose lives within the QVT specification. As the figure suggests, an alternative way to specify pieces of a transformation is to use black-box mappings. The idea is that, whenever writing a mapping (either operationally or declaratively) is too complex, a black-box implementation provided in other languages such as Java or C# can be defined and used from QVT-O or QVT-R transformations. We heavily rely on this feature to support the execution of our quality-driven model transformations, as it will be clear in

the next chapter while talking about execution.

4.2.1. The QVT-Relations Language

QVT-Relations is a declarative model-to-model transformation language with bidirectional capabilities. Declarative means that the transformation does not specify how elements in the source model should be translated into elements of the target model, but it specifies what constraints must be satisfied by the elements in the source and target models to be compliant with the transformation. Bidirectional means that the same transformation specification, being basically a set of constraints, may be used in both directions: from the source model to the target model, and the contrary. This is particularly useful in settings where the designer modifies the target model produced by a transformation (for example a lower abstraction model) and wants to push back the modifications also to the source model (for example an higher abstraction model).

Concisely, a transformation between two or more models (called **domains**) is represented by a set of **relation**s which specify the constraints that must hold between model objects. Domains specify what kind of models participate to the transformation: they are named and they determine the meta-model to which input/output models of the transformation must comply.

As it is evident from the name of the language, relations constitute the basic building block a QVT-R transformation. Relations declare the patterns and the constraints that must be satisfied by elements in the candidate models. A relation is constituted by three parts: the **pattern specification**, an optional **when** clause, and an optional **where** clause. In order to clarify all these concepts, let us consider the transformation fragment in Listing 4.1. The depicted fragment is extracted from the ORM example available in the QVT specification and deals with translating all the domain classes defined in a *Package* into the corresponding *Table*s of a database schema. The relation *ClassToTable* on lines 8-36 defines this mapping.

Lines 10-27 in the listing outline the structure of pattern specifications. Each relation must define a pattern for every domain involved in the transformation. The QVT specification provides the clearest definition of a domain: *"a graph of object nodes, their properties and association links originating from an instance of the domain's type"* [41]. Pattern specifications are expressed in terms of the domain meta-models, and they define the template to match, modify and create domain objects. For example, the pattern defined on lines 10-14 for the *uml* domain matches objects whose meta-type is *Class* and which are *persistent*. Lines

Listing 4.1: Anatomy of a relation.

```
1
top relation PackageToSchema                                    2
{                                                               3
  checkonly domain uml p:Package{...}                           4
  enforce domain rdbms s:Schema{...}                            5
}                                                               6
                                                                7
relation ClassToTable                                           8
{                                                               9
  checkonly domain uml c:Class {                                10
    namespace = p:Package {},                                   11
    kind = 'Persistent',                                        12
    name = cn                                                   13
  }                                                             14
                                                                15
  enforce domain rdbms t:Table {                                16
    schema = s:Schema {},                                       17
    name = cn,                                                  18
    column = cl:Column {                                        19
      name = cn+'_tid',                                         20
      type ='NUMBER'                                            21
    },                                                          22
    primaryKey = k:PrimaryKey {                                 23
      name = cn+'_pk',                                          24
      column = cl                                               25
    }                                                           26
  }                                                             27
                                                                28
  when {                                                        29
    PackageToSchema(p,s);                                       30
  }                                                             31
                                                                32
  where {                                                       33
    AttributeToColumn(c,t);                                     34
  }                                                             35
}                                                               36
                                                                37
relation AttributeToColumn {                                    38
  checkonly domain uml c:Class {...}                            39
  enforce domain rdbms t:Table {...}                            40
}                                                               41
```

16-27 define instead a pattern for the *rdbms* domain matching objects of type *Table* with a primary key column.

Patterns do not live on their own: they may declare variables, they may define how to bind variables (i.e., how to compute values for vari-

ables), and they may reference variables defined in other pattern specifications. Our listing shows also this feature. Line 13 defines a variable named *cn* whose value will correspond to the name of the matched *Class* object. The same variable is then used on line 18 to match only *Table*s whose name is equal to the value of the *cn* variable (*Table*s with the same name of the matched *Class* object), on line 20 and on line 24, to compute the name of the *Column* acting as record identifier and of the *PrimaryKey*, respectively. Another example is shown on lines 19 and 25, where the variable *cl* is used to link the *PrimaryKey* object with the *Column* matched/created to hold the value of the record identifier.

The optional when clause of a relation (lines 29-31) defines the precondition for the relation, that is, the conditions under which a relation must hold. The optional where clause (lines 33-35) defines instead the postcondition, or what conditions must hold when a relation holds. When and where clauses are constituted by a list of boolean predicates written with Object Constraint Language (OCL) [103] clauses. Predicates may define constraints either on the objects matched by template specifications or on other relations. The latter case is used to define precedence between relations, if the predicate belongs to the when clause, or a simple call-out semantics if the predicate belongs to the where clause. These two mechanisms represent a convenient way to modularize transformations.

In our reference example, the when clause establishes a precedence constraint with the *PackageToSchema* relation, that is, the *ClassToTable* relation may be invoked only after the invocation of the *PackageToSchema* relation. The constraint also imposes a binding for the *p* and *s* variables used in the pattern specification (defined in lines 11 and 17, respectively), that is, the *Package* and the *Schema* pointed by the *p* and by the *s* variables must be the same pointed by the already executed *PackageToSchema* relation. The where clause instead forces the invocation of the *AttributeToColumn* relation after the *ClassToTable* relation is executed. By doing so, it enforces the mapping of all the *Attributes* of the matched *Class* onto *Column*s of the matched *Table*.

Relations may be **top-level** or not. The most simple way to explain what is the difference between top-level and normal relations is to explain how Relational transformations are actually executed, or their execution semantics. QVT-Relations transformations can be executed in two different modes: a **check-only** mode and an **enforce** mode. In the check-only mode, only consistency is verified by checking that all constraints specified by the transformation hold. In the enforce mode, a direction must be specified and the target model pointed by the marked domain is modified in order to respect the constraints specified by the transformation relations. If a valid match for all the domain templates

of a given relation is found, the relation succeeds and nothing is done. Otherwise, if a match is not found for the enforce domain template, the objects in that domain are modified or new ones are created as defined by the relation. For both the execution modes, the execution semantics of relational transformations is similar to well-known textual grammars [100]. In fact, it has been proven that the Relations language is nearly equivalent to the TGGs formalism [104], an extension of textual grammars to graphs. Relations are executed by matching objects in the input models according to the corresponding pattern specifications, and by checking or enforcing the constraints on the output model as specified by the pattern specifications of the target domain. In this sense, top-level relations absolve to the same role of axioms in well-known textual grammars; they represent the starting point for the transformation execution and are required to hold. Non-top-level relations may be invoked by other relations, and are not required to hold for the transformation to succeed.

Defining Meta-class Identifiers

Another aspect of the QVT-Relations language that deserves to be described, given the extensive use we make in QVT-Rational, is the definition of keys for meta-classes. As we just showed, pattern specifications also serve the purpose of defining how objects can be created in the target domain model if they do not exist. For example, let us consider again the *ClassToTable* relation and let us consider the existence of the *Product* class in the source model. When that relation is executed, there is no valid match for the *rdbms* pattern specification: no *Table* named *Product* already exists, a new one is thus created. However, during the execution of the *AttributeToColumn* relation we expect to create as many database *Column*s as many *Attribute*s are defined for the *Product* class, and we expect that they are appended to the same *Product Table* created during the execution of the *ClassToTable* relation, not to a newly created one.

In order to ensure that duplicate objects are not created when required objects already exist, QVT-Relations provides a mechanism to specify how objects can be identified. This mechanism is based on the **key** construct, *"which defines a set of properties of a class that uniquely identify an object instance of the class in a model"* [41]. Keys are used at the time of object creation; if an object template expression has properties corresponding to a key of the associated class, then the key is used to locate a matching object in the model; a new object is created only when a matching object does not exist.

Listing 4.2 shows how the **key** construct can be used to define keys

for the meta-classes defined in the *Simple-UML Class* meta-model and in the *RDBMS* meta-model. For example, elements of of type *Package* will be identified only by the value of the *name* attribute. Elements of type *Classifier* (and of all its subtypes) will be instead identified by the value of the *name* and of the *namespace* attribute, that is, they will be identified also by the *Package* which encloses them.

Listing 4.2: An example of key definitions.

```
                                                                    1
-- Keys for the uml domain                                          2
key uml::Package {name};                                            3
key uml::Classifier {name, namespace};                              4
key uml::Attribute {name, owner};                                   5
key uml::PrimitiveDataType {name, namespace};                       6
                                                                    7
-- Keys for the rdbms domain                                        8
key rdbms::Schema {name};                                           9
key rdbms::Partition {name,schema};                                 10
key rdbms::Table {name, partitionId, schema};                       11
key rdbms::Column {name, owner};                                    12
key rdbms::Key {name, owner};                                       13
```

4.2.2. The QVT-Operational Language

QVT-Operational is an imperative-styled language supporting only unidirectional transformations. Imperative means that the syntax of the language is similar to well-known general purpose programming languages such as Java and C#. The transformation specifies how the translation between input/output models must be carried out by listing all the steps that must be executed. This is obviously in contrast with the other flavor for writing model transformations, i.e., the declarative style proposed by QVT-Relations. Unidirectional means instead that a transformation may be executed only in the direction for which it has been thought and developed, it cannot be used in the opposite direction or when modifications in the output models must be pushed to the input models of the transformation. To handle such situations, two model transformations must be written with opposite directions.

An operational transformation is defined by a **signature** — specifying the transformation direction, the models, and the meta-models involved in the mapping — and by a **main** procedure — a transformation entry point similar to the one programmers are used to define for standard applications. Listing 4.3 shows the signature and the main procedure for

the ORM running example. The transformation takes in input a *classModel* of type *simpleUml* and generates a *dbModel* of type *rdbms*. For each parameter specified in the signature a *direction kind* must be specified in order to determine how models participate to the transformation. Direction kinds are used also in mapping specifications as we will see in a short, and QVT-O distinguishes between three kinds of directions: **in** for models/elements that must be accessed in read-only mode, **out** for models/elements that will receive a new value during the execution, and **inout** for models/elements which retain their original value but that may be updated during the execution of the transformation.

Listing 4.3: A transformation signature and the associated main procedure.

```
1
-- The transformation signature                                    2
transformation Uml2Rdbms(                                          3
        in classModel:simpleUml,                                   4
        out dbModel:rdbms);                                        5
                                                                   6
-- The main procedure                                              7
main() {                                                           8
        var packages := classModel.rootObjects()[Package];         9
        assert fatal (packages->size() = 1)                        10
                with log ("Input model does not                    11
                        contain exactly one package.");            12
                                                                   13
        packages->map Package2Schema();                            14
}                                                                  15
```

Similarly to QVT-Relations where the constituents of a transformation were relations, for an operational transformation the constituents are **mapping**s. Mappings determine how elements of the source models are mapped onto elements of the target models. The structure of mapping operations is similar to the structure of a relation. Listing 4.4 shows the general structure of a mapping which is constituted by a signature, an optional guard (the **when** clause), an optional postcondition (the **where** clause), and a mapping **body** which absolves to the same purpose of pattern specifications for relations (i.e., it defines the instructions to perform the transformations but in an imperative manner).

Lines 1-3 depict the general structure of a mapping signature. Different types of mappings may be declared and are supported by QVT-Operational. When a meta-class is specified (the *X* token on Line 1) the mapping is said to be *contextual*, that is, the mapping is associated

Listing 4.4: Anatomy of an operational mapping.

```
mapping <dirkind> X::mappingname                                    1
  (<dirkind> p1:PT1, <dirkind> p2:PT2)                              2
  : r1:RT1, r2:RT2                                                  3
  when {...}                                                        4
  where {...}                                                       5
  {                                                                 6
    init {...}                                                      7
    population {...}                                                8
    end {...}                                                       9
  }                                                                 10
```

to a specific meta-type which will be implicitly passed as an argument during the invocation. This mechanism is similar to instance methods for Object Oriented programming languages. On the contrary, when the X token is missing the mapping is said to be *non-contextual*. Line 2 specifies instead the list of the mapping arguments, while Line 3 defines the output parameters. Both for mapping arguments and for the signature a direction kind may be specified, with the same meaning that we outlined when we described the signature of a transformation.

Lines 4 and 5 show instead the when and the where clauses, respectively. The when clause restricts the execution of the mapping, by specifying a set of boolean predicates over the elements involved in the mapping whose evaluation enables/disables the triggering of the mapping itself. The where clause works similarly to QVT-Relations, it may contain a set of instructions which are executed after the execution of the mapping body.

The body of a mapping contains the instructions to actually execute the translation and may contain several optional parts in addition to these instruction. In detail, the optional *init* section may contain a set of instruction which will be executed before instantiating the output parameters. This is useful when particular instructions must be executed prior to every other instruction defined in the mapping body and in the optional sections, for example, when particular conditions or assertions must be checked to signal potential error conditions. The init section is also useful to instantiate all the output parameters with values different from the default. The *population* section is executed after the *init* section and before every other instruction in the mapping body; as the name suggests, it serves the purpose of populating *out* and *inout* mapping parameters. The *end* section is instead the last executed section, and may contain any kind of instruction similarly to the mapping body.

Mapping Instructions

QVT-O proposes an imperative style to define how transformations should be carried out, and the language proposed to define the sequence of instructions to execute is based on the Object Constraint Language (OCL) standard [103].

OCL is an OMG proposal born to express, as it is clear from the acronym, constraints on models. It provides constructs to define complex query over model elements such as selection or projection. OCL queries can be thought as a set of instruction to build sets of model elements; OCL in fact provides a comprehensive list of operators for sets such as union or disjunction.

OCL however lacks appropriate constructs to express model transformations with an imperative style. The usual constructs available in general purpose programming languages to organize the computation in steps, such as branches or loops, are missing. It also misses the concepts to define variables, assignments, blocks of code, and computations. QVT-O extends OCL with all such constructs with the Imperative-OCL standard and a small library providing well-known data structures (e.g., maps, sequences, bags, and lists [105]) useful when writing imperative source code.

Mapping bodies can be consequently written with such standard, and an example is shown in Listing 4.5 taken from the QVT-O transformation we used to implement the ORM example. The depicted mapping is a contextual mapping associated to the *Class* meta-class whose task is translating a domain entity into a database table. Lines 4-6 define the *init* section and show the usage of the assignment operator introduced in Imperative-OCL. In this case, we assign to the result of the mapping the same table we received input. **Result** is a special keyword available in mapping operations defining a unique output variable, which points to the result of the mapping. Other two special variables are available in a mapping: the **this** variable pointing to the transformation object which may be used to access transformation properties, helpers and for reflection purposes, and the *self* variable which points to the model element assigned to the mapping context in case of a contextual mapping. Line 7 is another example of assignment, but in this case we use it to assign to the resulting database table the same name of the input domain entity. Line 10 shows another important construct added to Imperative-OCL, the **map** construct. The map keyword is used to invoke other mappings (similarly to standard method invocation for general purpose languages), but has the side-effect of eliciting this fact to the execution engine, with consequent creation of transformation traces. In the specific case of Line

10, we invoke the *Attribute2Column* mapping for every attribute defined in the input domain entity. Lines 13-20 show instead a more complex instruction block. Without digging too much into the details, it creates and assigns to the *hasKey* property of the resulting database table a new *Key* object, containing a reference to each column corresponding to an attribute of the domain entity marked as identifying. Line 15 is in charge of this, and shows the usage of the **forEach** construct to iterate over a collection of elements and execute a sequence of instructions for each of them.

Listing 4.5: An example of a QVT-O mapping body.

```
-- Base mapping for all ClassToTable mappings                          1
mapping Class::ClassToTableActual(inout table:Table) : Table           2
{                                                                      3
 init {                                                                4
  result := table;                                                     5
 }                                                                     6
 result.name := self.name;                                             7
                                                                       8
 -- Map all attributes                                                 9
 result.column += self.attribute->map Attribute2Column();              10
                                                                       11
 -- Map id attributes on autogen cols and primary key                  12
 table.hasKey :=  object Key {                                         13
  name := self.name + "_pkey";                                         14
  self.attribute->select(a | a.isId = true)->forEach(attr) {           15
   var mappingCol := attr.resolveoneIn(                                16
    Attribute::Attribute2Column).oclAsType(Column);                    17
   column += mappingCol;                                               18
  };                                                                   19
 };                                                                    20
}                                                                      21
```

Modularizing Transformations

Transformations may be quite complex and may require writing many source code lines. Maintainability and development would be rather complex without features to modularize transformations. QVT-O provides some features for this specific purpose, which we extensively use in the rest of this thesis and need hence an explanation.

The first mechanism proposed by QVT-O to modularize transformations is based on the concept of library, a collection of mappings and helper functions which may be externally defined, imported, and consequently re-used. How libraries and imports work is rather similar to

the same-named mechanism for general purpose languages. A peculiar characteristic of QVT-O is however the possibility to define libraries in a different language and integrate them in a seamless way with native QVT-O transformations. This is the way black-box transformations are handled in QVT-O. For example, as we will explain later in Chapter 5, we use this feature to enable the execution of QVT-Rational transformation based on the operational standard, by defining a black-box library written in Java and invoking its exported functionalities directly from a QVT-O transformation.

The other mechanism to manage complexity leverages the possibility to structure mappings in hierarchies. In particular, three are possible ways provided by QVT-O to combine mappings: inheritance, merging, and disjunction. Listing 4.6 shows the syntax to express how mappings can be combined. Inheritance can be specified via the **inherits** keyword and by specifying the list of inherited mappings. A mapping that has inherited mappings invokes first its *init* section, then invokes the inherited mappings, then executes the rest of its body. Merging can be instead specified via the **merge** keyword. In terms of execution semantics, a mapping which merges other mappings executes the merged mappings after its execution. Finally, disjunction can be expressed via the **disjunct** keyword. The execution semantics of this composition operator is interesting. A mapping operation defining a disjunction cannot define a body, and its invocation results in the execution of the first mapping listed in the disjunction whose precondition (i.e., the when clause) evaluates to true. As we will explain in the next section, we use both merging and disjunction to represent variabilities and variation points into QVT-Rational transformations.

Listing 4.6: Structuring and combining mappings.

```
mapping inout <contexttype>::<mappingname>                    1
  (<parameters>,) : <result_parameters>                       2
  inherits <rulerefs>, merges <rulerefs>,                     3
  disjuncts <rulerefs>,                                       4
  when {<exprs>} where { <exprs>}                             5
```

4.3. Defining Variability

QVT-Rational provides feedback in the form of complete system variants, and how design variants are generated is defined by specifying a non-injective model transformation with variabilities. In order to enable

such mechanism to define feedback rules, we extend the two languages we described in the previous sections (i.e., QVT-Relations and QVT-Operational) with appropriate constructs to elicit variabilities, variation points, and variants. We add new constructs to represent variability-related concepts for three purposes.

First of all, these constructs absolve to rational purposes, i.e., they make clearer in a transformation which are the possible source of variability, how variants are computed, and how they relate to quality-properties. This is important both from the domain expert perspective, to ease maintenance and development tasks, and from the designer perspective, to clearly share knowledge about the possible strategies to cope with quality-related issues. Without proper mechanisms to highlight such information into model transformations, the feedback rules, variabilities, and variation points would be hidden and burdened inside the source code.

The second reason for the need of introducing specific concepts to express feedback rules comes from the fact that the execution semantics of QVT does not deal well with non-injection. This problem is however not specific of only QVT, but also other model transformation languages suffer from it. The issue resides in the fact that either the model transformation language does not natively permit the expression of non-injective transformations (and thus of variabilities) or it does permit that but the execution semantics does not permit distinguishing between them or generating all the possible outputs. An example of the first case is QVT-Operational, and in general every language promoting an imperative style. With declarative languages such as QVT-Relations or TGGs, given their pattern matching semantics, it is relatively easy to express non-injective transformations: it is sufficient to write two transformation rules with overlapping Left Hand Side (LHS) patterns. However, the execution semantics of such languages usually states that the transformation succeeds when a model compliant with the constraints expressed in the transformation is found; we instead need to be able to potentially generate all the possible candidate output models and distinguish them according to the exhibited quality properties.

The third and last reason comes from the necessity of expressing some concerns which are not entirely present into model transformation languages. The most evident concern in this sense is quality, which is also the reason motivating the introduction of quality-driven model transformation languages. To be able to generate system variants and feedback about quality, we need to bind variability to quality and to specify how quality can be predicted. Without proper constructs, expressing this kind of information in *vanilla* model transformation languages would be

impossible.

In the rest of this section we describe the constructs we introduce to elicit variability and variation points, and we show how they can be used in QVT-O and in QVT-R to define the feedback rules. We indeed clarify here that, even if all the concept we present are concretized in the context of QVT, they are quite general and may be easily extended to other model transformation languages.

4.3.1. The Rational Annotations

To elicit feedback rules into model transformations QVT-Rational provides a set of annotations similar to those available in other general purpose programming languages (i.e., the code annotations a-la-Java or the attributes a-la-C#). The standard way to add new concepts to a language would be to modify its grammar with the rules to recognize the new constructs. This however brings as a consequence several disadvantages. When the grammar of a language is modified, all the artifacts depending on it (editors, compilers, virtual machines, execution engines) must also be modified accordingly. Using annotations mitigates these kind of implementation-related issues, reducing thus the effort of maintaining QVT-Rational over time and the effort of extending our approach to other model transformation languages in future. Annotations may in fact be stripped out before concrete execution (i.e., no heavy modifications to the execution engines must be applied) and, by placing them inside comments, editors do not need to be changed at all.

The new concepts provided by QVT-Rational are shown in Figure 4.3, while a brief description of the annotations is provided in Table 4.1. Listing 4.7 shows instead the grammar of the annotations with the Extended Backus-Naur Form (EBNF) notation. Four are the annotations we provide. In the following we provide a brief description of them and we anticipate some of the detail which we cover extensively in the next sections.

Two are the most important annotations we provide, the *@varpoint* and the *@variant* annotation. The former serves the purpose of marking the spots (i.e., the code locations) in the transformation source code responsible for declaring a variation point. The latter serves instead the purpose of annotating in the transformation source code the locations where variants are declared and defined. In the case of relations-based QVT-Rational transformations, the @varpoint annotation is used to mark the predicates in where clauses which list the possible variants for a variation point, while the @variant annotation is used to mark the relations which implement the viable variants. Similarly, for QVT-O

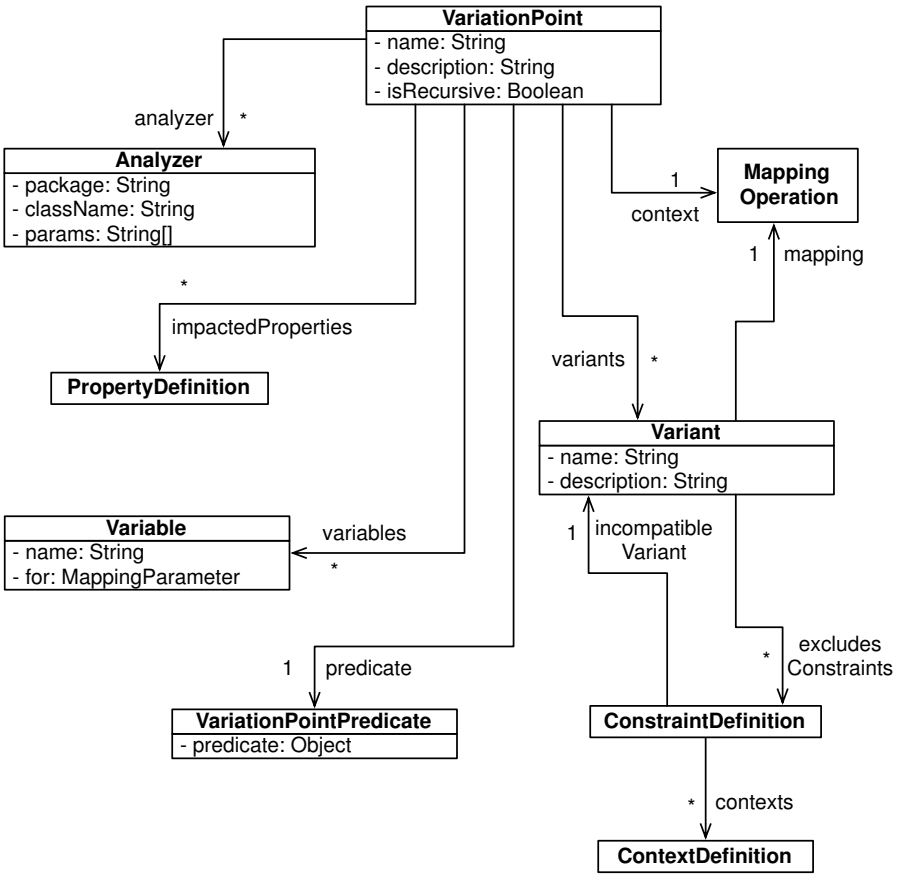

Figure 4.3.: The *Variability* meta-model.

transformations the @varpoint annotation is used to mark disjunctive mappings which list the viable alternatives, while the @variant annotation is prepended to the operational mappings defining how variants can be generated. This aspects will be clarified in the next sections, where we concretely show how variants and variation points can be specified.

Besides marking the code locations important from the perspective of feedback generation, annotations also carry extra information. The *name* and *description* attributes are used for documentation purposes. The *name* attribute also helps in uniquely identifying variabilities and variants; this is used mainly for cross-referencing for example in the definition of constraints between alternatives. The *impact* and *analyzer* attributes carry instead information about quality and bind variation points to it. The *impact* attributes defines the list of non-functional prop-

Annotation	Purpose	Properties	
		Name	Purpose
@varpoint	Marks the point in a transformation where a variability occurs and binds it to quality.	name	Uniquely identifies the variation point.
		description	Describes the variability.
		subject	Explicitly marks the domain entities participating to the variability.
		impact	Binds the variability to the quality properties.
		analyzer	Specifies how to predict quality
@variant	Marks the point in a transformation which specifies a variant.	name	Uniquely identifies the variant.
		description	Describes the variant.
		requires	Lists the variants required by this variant.
		excludes	Lists the variants excluded by this variant.
@keydefs	Lists the metaclass keys. Used when not natively supported by the transformation language.		
@nfpmodel	Used to import a quality model.		

Table 4.1.: The list of annotation provided by QVT-Rational.

erties impacted by a specific variability, while the *analyzer* attributes specifies how quality properties can be evaluated by defining the MDQP tool-chain to invoke. The *subject* attributes serves the purpose of explicitly specifying which modeling entities participate to the variability. We recap from Section 3.2.1 that variabilities are in many cases model-

Listing 4.7: The grammar of the annotations provided by QVT-Rational.

```
<varpoint> ::= '@varpoint' '{'
                  'name' ':=' <id>
                  'description' ':=' <string>
                  (<subject_def>)?
                  <analyzer_def>
                  <impact_list>
               '}'
<subject_def> ::= 'subjects' ':=' '{' <id> (',' <id>)* '}'
<feature>  ::= <meta_class_id> '::' <feature_id>
<analyzer_def> ::= 'analyzer' ':=' <class_name>
<class_name>   ::= <id> ('.' <id>)*
<impact_list> ::= 'impact' ':='
                  '{' <impact_def> (',' <impact_def>)* '}'
<impact_def>    ::= <quality_prop> '(' (<arg> (',' <arg>))? ')'
<quality_prop>  ::= <model_id> '::' <prop_id>
<arg>           ::= <catch_all> | <string> | '$'<ocl_query>
<catch_all>     ::= '_'

<model_id>      ::= <id>
<meta_class_id> ::= <id>
<feature_id>    ::= <id>
<prop_id>       ::= <id>

<variant> ::= '@variant' '{'
                  'name' ':=' <id>
                  'description' ':=' <quoted_string>
                  <requires>
                  <exludes>
               '}'
<requires> ::= 'requires' ':='
               '{' <require> (',' <require>)* '}'
<require>  ::= <id> '('(require_arg (',' require_arg)*)?')'
<excludes> ::= 'excludes' ':='
               '{' <exclude> (',' <exclude>)* '}'
<exclude>  ::= <id> '('(arg (',' arg)*)?')'
<require_arg> ::= <string> | '$'<ocl_query>

<keydefs> ::= '@keydefs' ':='
              '{' <keydef> (',' <keydef>)* '}'
<keydef> ::= <model_id> '::' <feature_id>
             '(' '#'<feature_id> (',' '#'<feature_id>)* ')'

<nfp_model_import> ::= '@nfpmodel' [<alias> '=>' <location>]
<alias>             ::= <id>
<location>          ::= string
```

dependent, that is, the selection between variants depends on information contained into source models such as the existence of particular objects. The subject attribute is useful when extra parameters are passed to relations or mappings to simplify their specification, but which do not participate concretely to the variability. The *requires* and *excludes* instead permit the specification of constraints between variants.

Concerning the other two annotations, the *@keydef* annotation provides a convenient mechanism to define meta-class keys when the extended model transformation language does not natively support that feature. The *@nfpmodel* annotation is instead used to declare which quality properties are involved in the feedback process, by defining which quality model should be imported for usage in the transformation.

4.3.2. Expressing Variability With QVT-Relations

We already anticipated that QVT-Relations is natively capable of expressing non-injective transformations with alternatives. However, we have not yet shown how to concretely express them in terms of relations. Given a specific variability, variants can be expressed through different relations, while variation points correspond to boolean predicates over the possible alternatives, in the post-condition of a relation. Both model dependent and model independent variabilities can be expressed in this manner. The only difference is the presence of an *absolute constraint* to enforce that the same alternative is consistently selected while transforming; this detail will be clarified later in Section 4.5 while talking about constraints and annotations.

The transformation fragment in Listing 4.8 concretely shows how we express variabilities, in the specific case of the inheritance mapping variability for the ORM example. The boolean predicate on Lines 10-11 is tagged with the @varpoint annotation and lists all the viable alternatives to map inheritance hierarchies. The relations on lines 20 and 30 which are annotated with the @variant marker define instead how the two strategies are implemented.

Before proceeding, we must indeed point out that this way of expressing variability is just one of the possibilities, other techniques are available to express variants and variation points in QVT-Relations. For example, Insfran et al. in [18] use different top-level relations with overlapping source domains to represent variants. This technique however introduces some issues from an implementation perspective. As we will see later in Chapter 5, we need to intercept when variation points are encountered during a transformation. Using top-level relations would have required modifying existing interpreters to control the order in which

Listing 4.8: The inheritance mapping variability with QVT-R.

```
top relation SubClassToTable {
 checkonly domain uml c:Class { ... };
 enforce domain rdbms superTable:Table { ... };
 ...
 where {
  @varpoint {
   name := InheritanceStrategy
   ...
  }
  SubClassToParentTable(c, superTable) xor
    SubClassToTableAndAssoc(c, superTable);
  ...
 }
}

@variant {
 name := UpwardCollapse
 ...
}
relation SubClassToParentTable {
 checkonly domain uml c:Class { ... };
 enforce domain rdbms superTable:Table { ... };
 ...
}

@variant {
 name := AssociationCollapse
 ...
}
relation SubClassToTableAndAssoc {
 ...
 checkonly domain uml c:Class { ... };
 enforce domain rdbms superTable:Table { ... };
 ...
}
```

top-level relations are executed, which would have required non-trivial changes to existing interpreters and conflicts with our goal of being implementation agnostic as much as possible.

To be sound, relations and predicates used to express variabilities must satisfy some conceptual and syntactic criteria. We said at the beginning of this section that alternatives are defined by relations, and each relation is composed of several domains. More formally, we require that

Rule 1 *Given a specific variability, the signatures of the relations implementing its variants must be compatible according to the execution*

direction.

The execution semantics of QVT-Relations requires the domain whose model will be created from scratch or checked for consistency to be marked as the target. The remaining domains are only an input for the transformation, they specify the context of a variability, and we require them to be compatible. Consider the transformation fragment shown in Listing 4.8 and suppose that the target domain is *rdbms*. Two variants are viable for the variation point defined in lines 10-11 (the *Upward-Collapse* strategy and the *AssociationCollapse* strategy), both relations implementing them accept the same type of objects for the remaining *uml* domain.

We also distinguish between mutually exclusive, optional and recursive variation points. According to this classification, the predicates defining a variation point must comply with the following conditions. Before proceeding, we need to recall that boolean operators in the Relations language are short-circuited, as it happens for many programming languages. For example, in case of an *or* predicate, the evaluation to true of a term in the predicate is sufficient to let the whole predicate evaluate to true, and the evaluation of the subsequent terms is skipped. The following definitions work under this assumption.

Rule 2 *Mutually exclusive variation points must be expressed through predicates* xor*-ing the possible alternatives, in the other case (non-mutually exclusive variation points) the* or *operator must be used.*

Rule 3 *Optional variation points are non-mutually exclusive variation points,* or*-ing a unique variant with the* true *boolean value.*

Rule 4 *Recursive variation points are optional variation points, and the alternative must be the same relation in which the variation point resides.*

The variation point shown in Listing 4.8 on lines 10-11 is an example of mutually exclusive variation point, in fact we use the *xor* operator. Listing 4.9, which depicts how we can implement the partitioning variability for the ORM running example, shows an example of an optional recursive variation point, which is useful in situations when a modeling object may be mapped several times onto the same type of objects.

In particular, the definition of the partitioning variability requires the usage of two variation points. First a mutually exclusive variation point (Lines 7-8) deciding about whether partitioning should be used or not to map a domain entity in a database schema, and a second optional recursive variation point (Lines 34-35) deciding about how many slices

should be put inside the partition. A special mention needs the recursive variation point. By looking at the transformation it is clear why we use the term recursive, the *ClassToAnotherTableInPartition* relation recursively invokes itself several times, each time incrementing the value of the *sliceId* domain. This recursive variability also shows the usage of the subject annotation on Line 32, where we specify that only the *c* and the *p* domain (pointing respectively to the domain entity being translated and to database table used for the mapping) should be considered as parameters to the variability, while the *sliceId* domain should be excluded.

The interested reader can find the entire transformation (showing the complete annotations, the complete mapping relations, and all the details) in Appendix A.

Listing 4.9: The partitioning variability with QVT-R.

```
1  relation ClassToTable
2  {
3    domain uml c:Class {...}
4    domain rdbms t:Table {...}
5    where {
6      @varpoint{ name := UsePartitioning }
7      ClassToSingleTable(c,t)
8        xor ClassToTableInPartition(c,t);
9    }
10 }
11
12 @variant{ name := ClassToPartition }
13 relation ClassToTableInPartition
14 {
15   domain uml c:Class {...}
16   domain rdbms t:Table {...}
17   where {
18     ClassToAnotherTableInPartition(c,
19       t.partition, 1);
20   }
21 }
22
23 @variant{ name := AnotherSlice }
24 relation ClassToAnotherTableInPartition
25 {
26   domain uml c:Class {...}
27   domain rdbms p:Partition {...}
28   primitive domain sliceId : Integer;
29   where {
30     @varpoint{
31      name := NumberOfSlices,
32      subjects := {c,p}
33     }
34     ClassToAnotherTableInPartition(c,
35       t.partition, sliceId + 1) or true;
36   }
37 }
38
39 @variant{ name := SingleTable }
40 relation ClassToSingleTable
41 {
42   domain uml c:Class {...}
43   domain rdbms t:Table {...}
44 }
```

4.3.3. Expressing Variability With QVT-Operational

Differently from its declarative counterpart, QVT-Operational does not natively support variabilities and non-injective transformations. Its execution semantics precludes this possibility. Nonetheless, by using our annotations and, as we will see in the next chapter, thanks to our extended execution model it is indeed possible to express variabilities (and thus feedback rules) also in QVT-Operational.

The way we express variabilities with this operational transformation language is similar to what we have seen for QVT-Relations. Given a specific variability, variants can be expressed through different operational mappings, while variation points correspond to mapping operations listing all the possible alternatives in their signature through disjunction or merging.

Also in the case of QVT-O how variabilities are expressed must obey to some syntactic and correctness criteria. More formally, we require that

Rule 5 *Given a specific variability, the signatures of the relations implementing its variants must be compatible.*

This criterion is similar to the same one we stated for QVT-R, with the exception that now there is not anymore the concept of direction. As a consequence, the criterion translates to having signatures for the mapping operations implementing the alternatives with the same kind of input and output parameters.

Also in the case of QVT-O we also distinguish between mutually exclusive, optional and recursive variation points. The only matter that changes between the two languages is how variation points are expressed. In detail, we require that

Rule 6 *Mutually exclusive variation points must be expressed through an operational mapping listing all the possible variants with a disjunction. In the other case (non-mutually exclusive variation points) the* merge *operator must be used.*

Rule 7 *Optional variation points are non-mutually exclusive variation points with a unique variant listed in the merge predicate.*

Rule 8 *Recursive variation points are optional variation points, and the alternative must be the same relation in which the variation point resides.*

Listings 4.10 and 4.11 contain the transformation fragments relevant to show how we express the inheritance mapping and the partitioning variability for the ORM example, respectively.

Listing 4.10: The inheritance mapping variability with QVT-O.

```
@varpoint { name := InheritanceStrategy }
mapping Class::InheritanceMapping_VariationPoint() : Table
  disjuncts Class::AssociationCollapse_Variant,
  Class::UpwardCollapse_Variant;

@variant { name := UpwardCollapse }
mapping Class::UpwardCollapse_Variant() : Table {
 ...
}

@variant { name := AssociationCollapse }
mapping Class::AssociationCollapse_Variant() : Table
{
 ...
}
```

Listing 4.11: The partitioning variability with QVT-O.

```
@varpoint { name := UsePartitioning }
mapping Class::UsePartitioning_VariationPoint()
  disjuncts Class::ClassToSingleTable_Variant,
  Class::ClassToPartition_Variant;

@variant { name := ClassToSingleTable }
mapping Class::ClassToSingleTable_Variant() {
 ...
}

@variant { name := ClassToPartition }
mapping Class::ClassToPartition_Variant() {
 ...
 -- Check if more slices should be added
 self.map AddAnotherSlice_VariationPoint(partition.size + 1);
}

@varpoint { name := SlicesNumber, subjects := {self} }
mapping Class::AddAnotherSlice_VariationPoint(
  in sliceId:Integer)
  merges Class::AnotherSlice_Variant;

@variant { name := AnotherSlice }
mapping Class::AnotherSlice_Variant(in sliceId:Integer) {
 ...
 self.map AddAnotherSlice_VariationPoint(sliceId + 1);
}
```

For each variation point, all the listed variants respect Rule 5, they in fact exhibit compatible signatures from the point of view of the input and of the output parameters. In both cases, the kind of variability that we define is a mutually exclusive variability, and we in fact use the disjunct operator in the signature of the operational mapping implementing the variation points. As we did for the partitioning variability in the QVT-R case, also for the operational transformation we use two variation points and recursion. The variation point defined on Lines 1-4 of the second listing decides whether partitioning should be used or not to map a domain entity, while the variation point on Lines 18-21 decides about the number of slices to add to the partition. The recursive variability *SlicesNumber* is invoked by two code locations: first on Line 15 and then recursively on Line 26. As we did for the QVT-R case, for the *SlicesNumber* variability we also explicitly define the context of the variability by means of the *subject* annotation. In this case, we include only the domain entity being mapped. We use the *self* variable to indicate this since the domain entity being mapped is implicitly passed as the mapping context and not via input arguments.

The careful reader should have noted also that all the operational mappings depicted in Listing 4.11 do not list in their signature any reference to the output database table on which the input domain entity is mapped. We did this on purpose to simplify the writing of the transformation and a reference to the database table/partition onto which a domain entity is mapped can be retrieved by using the **resolution** operators offered by QVT-O. This feature is shown below in Listing 4.12, where we use the **resolveoneIn** operator on Lines 4-5 to retrieve the database table onto which the domain entity currently being mapped has been translated by the *ClassToTableMarker* mapping.

Listing 4.12: The body of the ClassToSingleTable_Variant mapping.

```
1 mapping Class::ClassToSingleTable_Variant() {
2  log("Mapping class " + self.name + " to single table");
3
4  var mappingTable := self.resolveoneIn(
5    Class::ClassToTableMarker).oclAsType(Table);
6  mappingTable.partitionId := 0;
7 }
```

The interested reader can find the entire transformation (showing the complete annotations, the complete mapping operations and bodies, and all the details) in Appendix A.

Defining Meta-class Keys

In the previous sections we outlined how meta-class keys can be defined in Relational transformations and we anticipated that they are important for our approach in order to be able to execute quality-driven model transformations. When a transformation language lacks the concept of meta-class keys and the corresponding constructs to define them, the *@keydef* annotation provided QVT-Rational can be used to overcome this limitation. This is the case of Operational transformations, where our annotation must be used to define keys. Listing 4.13 shows how the *@keydef* annotation can be used to define the same keys defined for ORM transformation expressed through QVT-Relations. The syntax is rather similar to the one proposed by the QVT-R language, first a pointer to the meta-class for which we are defining keys is given, then the set of attributes and features of the class which will act as keys are listed. Let us consider Line 2, the definition specifies that the *Package* meta-class will be identified by its *name* attribute. It is also possible to use references to other model elements to identify object of a particular type. Let us consider lines 3-4, both the *Class* and the *Classifier* meta-types are identified by their *name* attribute and by *Package* object to which they belong (pointed by the *namespace* reference).

Listing 4.13: Defining meta-class keys with QVT-O.

```
1  @keydefs{
2   simpleUml::Package(#name),
3   simpleUml::Classifier(#name, #namespace),
4   simpleUml::Class(#name, #namespace),
5   simpleUml::Attribute(#name, #owner),
6   simpleUml::Association(#name, #owner),
7   simpleUml::Generalization(#name, #owner),
8
9   simpleDb::Schema(#name),
10  simpleDb::Partition(#name, #schema),
11  simpleDb::Table(#name, #schema),
12  simpleDb::Column(#name, #owner),
13  simpleDb::Key(#name, #owner),
14  simpleDb::ForeignKey(#name, #owner)
15 }
```

4.4. Binding to Quality

So far we have only seen how variabilities can be defined in a model transformation written either in QVT-Relations or in QVT-Operational,

and how they can be marked with our specific annotations. We still need to show how variabilities can be bound to quality attributes, thus making our transformations quality-driven model transformations and enabling us to generate feedback about quality.

Binding to quality requires, from the perspective of the domain expert, the accomplishment of two tasks. First the quality metrics of interest for a particular engineering domain must be defined then, once they are available into QVT-Rational, information about the impact that each variability has onto each quality property must be included in the annotations of our extended model transformation language.

4.4.1. Defining Quality

QVT-Rational provides its own language to define quality attributes. The *NFPs* meta-model defines the provided concepts and is shown in Figure 4.4 and in Figure 4.5.

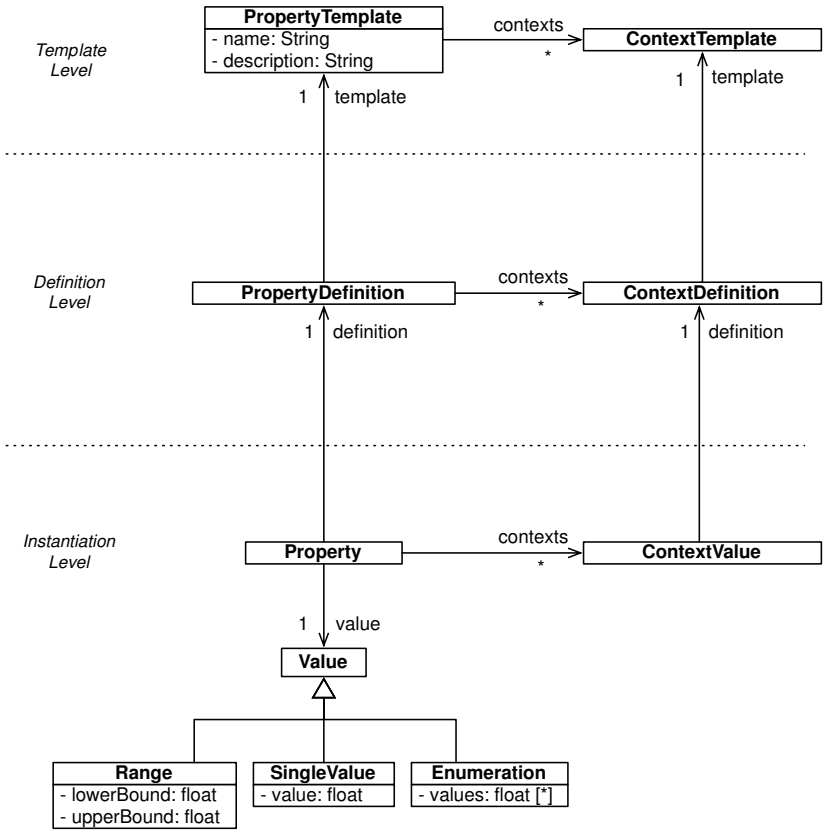

Figure 4.4.: The *NFPs* meta-model.

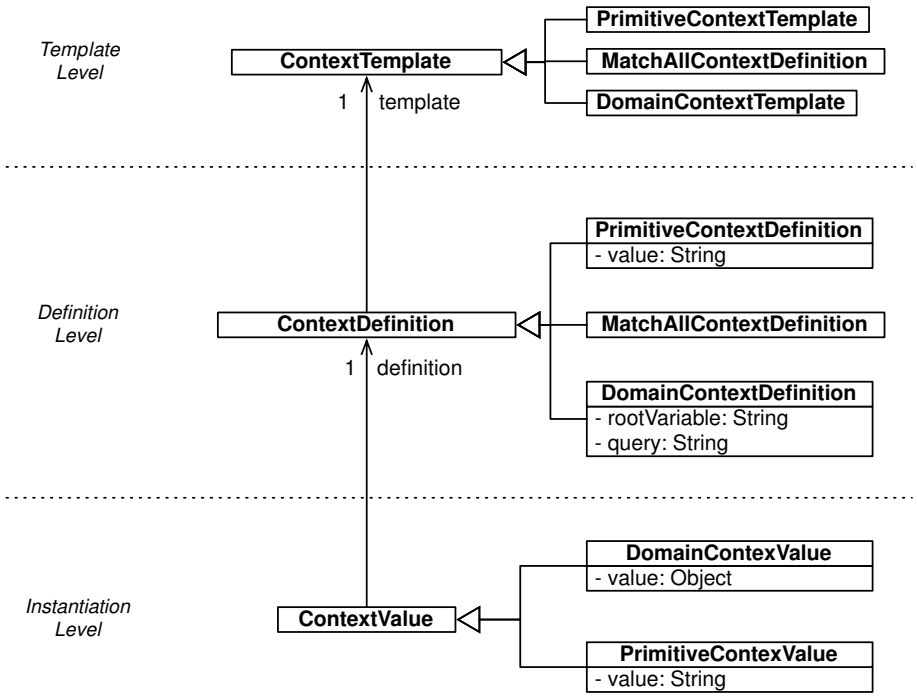

Figure 4.5.: Context-related meta-classes for the *NFPs* meta-model.

Quality metrics are usually not absolute, but they are measured with respect to particular system artifacts (what we call *context*s). Let us consider again the ORM example, given a class model and its corresponding database schema, different response times can be computed, depending on the domain entity we consider or the kind of query we are interested in. This characteristic is of primary importance to provide a powerful language able to define also non-trivial quality metrics, and it is reflected in the *NFPs* meta-model by introducing the concept of context and by splitting it into three parts: the *template* level, the *definition* level, and the *instantiation* level. Domain experts work with the concepts provided at the template level to define quality metrics, the definition level provides the concepts to bind the quality metrics defined at the template level to variabilities, while the instantiation level provides the meta-types to instantiate concrete quality metrics (i.e., the metrics and the values predicted for a particular model which will be shown to the end-user).

*Template*s specify properties in a generic manner: they act as prototypes to define non-functional properties in a way similar to how generics work in modern programming languages. A *PropertyTemplate* element corresponds to the definition of a quality metric and declares the types

of contexts with respect to which it is computed. To clarify the concepts of *PropertyTemplates* and of *ContextTemplates* let us consider again the ORM running example. In Section 3.3 we said that we are interested in computing two quality metrics, the performance of queries executed against the database schema (by measuring the response time of each query) and the wasted space. Wasted space and query response time are not absolute metrics, but they depend onto the domain entity being considered and also onto the particular query in case of the response time. Listing 4.14 shows how these quality metrics can be defined using *PropertyTemplates* (in the specific case, Listing 4.14 shows the textual syntax we have developed for the *NFPs* meta-model).

Listing 4.14: The quality metrics for the ORM case study.

```
1  import umlMM : "SimpleUmlMM.ecore";
2
3  PropertyModel Uml2Rdbms {
4          template GlobalResponseTime {
5                  context Query;
6          }
7
8          template ResponseTime {
9                  context Query;
10                 context Clazz[umlMM.Class];
11         }
12
13         template WastedSpace {
14                 context Clazz[umlMM.Class]
15         }
16 }
```

The import statement on Line 1 serves the purpose of importing the *SimpleUML* meta-model in order to use the concepts defined in it for the definition of *PropertyTemplate*s (we need in fact to reference the *Class* meta-type). Lines 4-6 define the *GlobalResponseTime* metric which is computed with respect to queries, while Lines 8-11 define the *ResponseTime* metric which highlights the contribution of each domain entity (i.e., the *Clazz* context) to the response time of each query. Lines 13-15 define instead the *WastedSpace* metric, which depends only on the considered domain entity.

The careful reader should have noted that we support two kind of contexts (i.e., primitive and domain contexts), and this is reflected also in the textual notation for the *NFPs* meta-model. *DomainContext*s are used to define contexts which reference meta-model concepts. In our example, for those quality metrics computed with respect to domain entities, we have the concept of *Clazz* defined in the *SimpleUml* meta-

model and we may reference it. By specifying the type of objects that can be bound to a quality property, we are able to perform some correctness checks when binding quality to variability, as we will see in a short. When no concept to which a quality metric needs to refer is formally defined in the MDE environment, *PrimitiveContext*s can be defined. In our example, this case is shown by the *Query* context, for which no corresponding formal concept is available in any meta-model defined for the ORM running example. Primitive contexts have the advantage of giving to domain experts the freedom to define any kind of context for quality metrics, but on the counterpart they disable any kind of type checking while binding to variability. Their value will be in fact a standard string, whose interpretation is left to tools and to humans without any kind of aid from our side.

4.4.2. Defining Impact

The property templates we just defined specify what are the quality properties and which kind of contexts they are computed with respect to. Two points that still are missing are how values for those contexts may be computed and how those quality metrics are affected by the variabilities defined in the model transformation. This information is crucial in order to generate feedback, we in fact need to understand what happens to the quality of a system when a particular variant is picked up instead of another in order to generate feedback.

Binding quality properties to variabilities can be accomplished by using the *impact* attribute of the *@varpoint* annotation. As we will see in a short, this also suffices to specify how values for *ContextTemplates* can be computed and the middle layer of the *NFPs* meta-model populated.

In order to define such binding, first the qualities defined by the domain expert must be imported into the quality-driven model transformation. This is accomplished by using the *@nfpmodel* annotation. An example of this is shown in Listing 4.15 taken from the ORM example, where the quality model we just defined is imported into the transformation to make available the concepts of response time and wasted space for usage in the annotations.

Listing 4.15: Importing quality metrics into a transformation.

```
1 @nfpmodel[nfps => "Uml2Rdbms.nfps"]
```

The following step consists in concretely specifying the binding between the imported quality metrics and the variabilities defined in the

transformation. As we mentioned before, this is accomplished by listing in the *impact* attribute for each variation point the list of affected metrics and how values for the contexts may be computed. Listing 4.16 shows how the binding is concretely defined in the specific case of the partitioning and of the generalization mapping variabilities for the QVT-O-based transformation. The very same syntax is used also for the Relational transformation, which we omit here for the sake of clarity.

Listing 4.16: Binding quality to variabilities via the *impact* attribute.

```
1  @varpoint {
2   name := InheritanceStrategy,
3   analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
4   impact := {
5    nfps::GlobalResponseTime(_),
6    nfps::ResponseTime(_, $"self"),
7    nfps::WastedSpace($"self")
8   }
9  }
10 mapping Class::InheritanceMapping_VariationPoint() : Table
11   disjuncts Class::AssociationCollapse_Variant,
12     Class::UpwardCollapse_Variant;
13
14 @varpoint {
15  name := UsePartitioning,
16  analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
17  impact := {
18   nfps::GlobalResponseTime(_),
19   nfps::ResponseTime(_, $"self")
20  }
21 }
22 mapping Class::UsePartitioning_VariationPoint()
23   disjuncts Class::ClassToSingleTable_Variant,
24     Class::ClassToPartition_Variant;
25
26 @varpoint {
27  name := SlicesNumber,
28  description := "Selects the number of slices",
29  subjects := {#self},
30  analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
31  impact := {
32   nfps::GlobalResponseTime(_),
33   nfps::ResponseTime(_, $"self")
34  }
35 }
36 mapping Class::AddAnotherSlice_VariationPoint(
37   in sliceId:Integer)
38   disjuncts Class::AnotherSlice_Variant;
```

Each quality metric listed in the *impact* attribute is accompanied by a specification of how values for contexts can be retrieved. Values for primitive contexts can be defined with a textual representation; we recall that primitive contexts have a string value. To specify how values for domain contexts can be retrieved OCL queries are used. Queries can be arbitrarily complex but must follow accessibility rules: the root object of the query must be visible in the context of the variation point, i.e., it is one of the objects pointed by the domain templates of the relation. For example, Line 7 in Listing 4.16 specifies that the *InheritanceStrategy* variation point affects the *WastedSpace* metric computed with respect to the domain entity pointed by the **self** variable. The same query is specified for the *ResponseTime* and *GlobalResponseTime* in the annotations for the other variation points.

The careful reader should have noted that for the other contexts we do provide neither an OCL query nor a textual value. We instead use another option to specify values for both primitive and domain contexts, the **catch-all** operator (i.e., the underscore). The catch-all works differently according to the kind of context for which it is used. In the case of domain contexts, it corresponds to selecting any modeling element compliant with the meta-type defined for the context at the template level. For example, let us suppose that we used the catch-all operator in Line 7 instead of the OCL query. It would correspond to any domain entity (i.e., any object of meta-type *Class* defined in the input model of the transformation) and it would mean that the variation point impacts the *WastedSpace* for every domain entity.

In the case of primitive contexts, its meaning is similar but our framework is not able to automatically derive the list of values compatible with the context. Its resolution is instead delegated to the analysis tool-chain used to predict quality, which has thus also the responsibility of defining the range of primitive contexts. Let us consider again our listing. Line 5 makes use of the catch-all operator for a primitive context and, from an abstract point of view, its meaning is that the variation point affects the response time of every query. Which values will be assigned to the context will be decided by the MDQP analyzer, who has the knowledge about which queries must be taken into account to predict the performance of the database schema and which domain entities they access.

Candidates will be evaluated, and feedback consequently generated, by predicting values for the properties listed in the *impact* attribute. However, estimation of non-functional properties is a domain specific matter: each application domain adopts particular meta-models, uses different transformations to automate the development process, and leverages different quality analysis techniques. We provide the *analyzer* annotation

to specify which analysis technique should be invoked for each particular variation point. In the next chapter we shall see that QVT-Rational has been implemented in the Java language, as a consequence, also analysis methodologies must be implemented with Java for interoperability. The fully qualified name of the class implementing the chosen technique constitutes the value of the *analyzer* attribute. If multiple analyzers are necessary, each one evaluating different properties, a comma separated list of classes can be specified. By default, three pieces of information are made visible to analyzers: the variation point being explored, the context in which it occurred (the model objects involved, the impacted properties, and transformation traces), and a reference to the usage model specifying the conditions under which system properties should be evaluated. If extra arguments are needed by a particular implementation, their value can be specified in the annotation using a method invocation-like syntax. In Listing 4.16 we use the *analyzer* attributes for every variation point: the *QnAnalyzer* class defined in the ORM package implements the analysis technique outlined in Section 3.3, and it is invoked to estimate impact on the response time and wasted space properties.

4.5. Constraining Variants

When multiple solutions are available to cope with the same problem (in our case a problem concerning quality) usually happens that different solutions for different problems may conflict. Our ORM running example contains an example of such conflicts: since data is usually assigned to the different slices of a partition according to the value of record identifiers, identifiers need to be managed by the underlying engine with appropriate algorithms; hence partitioning conflicts with the adoption of the random strategy for the identifiers generation variability.

QVT-Rational takes care also of this aspect and provides the *requires* and *excludes* attributes for the *@variant* annotation to specify positive and negative constraints, respectively. Negative constraints are defined by referencing the name of the conflicting variant in the *excludes* list. Two types of constraints are supported: absolute and contextual. Besides referencing the name of the conflicting variant, *contextual constraints* also specify — via OCL queries as we did for the *impact* list — which model entities should be involved in the conflicting variant for the constraint to be violated. On the contrary, *absolute constraints* use the special *underscore* keyword to match anything, and apply regardless of the entities participating in the conflicting variant. An example of negative constraint is shown in Listing 4.17 on Line 4, where we declare

that applying the randomized strategy to generate record identifiers conflicts with using partitioning for the same domain entity being mapped. Another example of negative constraint is shown on Lines 13-16, where we specify that the same policy to map generalizations must be applied consistently to every domain entity participating to the same inheritance hierarchy.

Listing 4.17: Specifying negative and positive constraints.

```
1  @variant {
2   name := RandomizedGeneration,
3   description := "A randomized generation strategy.",
4   excludes := ClassToPartition($"self.owner")
5  }
6  mapping Attribute::IdAttribute2AutogenRandomColumn_Variant()
7     : Column
8  { ... }
9
10 @variant {
11  name := UpwardCollapse,
12  description := "Maps a subclass on the parent table.",
13  excludes := AssociationCollapse($"self.namespace.generalization
14   ->select(g | g.superType = self.namespace.generalization
15    ->select(g1 | g1.subType = self)->any(true).superType)
16   .subType")
17 }
18 mapping Class::UpwardCollapse_Variant() : Table { ... }
```

Positive constraints are defined similarly, with the additional requirement that the *underscore* keyword cannot be used. All the model entities participating in the required variant must in fact be bound in order to enforce the selection of the required alternative.

4.6. Summary

In this chapter we showed how quality-driven model transformation may be defined in order to specify the strategies to cope with quality-related issues and to generate feedback in the form of complete system variants. We briefly described two model transformation languages we selected to specify quality-driven model transformation and we introduced the rational annotations that can be used by domain experts to mark variabilities, variants, and bind them to quality.

We must indeed discuss here two critical points of our approach:

- **Impact of the domain expert in the process.** The domain

expert is responsible for specifying how feedback can be computed by defining in a model transformation the various strategies to cope with quality-issues. The quality and the completeness of such information is essential for the success of our approach and for the quality of the feedback suggested by our system. The more strategies are formalized and the more complete are the feedback rules, the most precise and useful is the feedback. However, although the domain expert is, in our opinion, the best source for such kind of information, specifying feedback rules is a really difficult task. One of the most important (and difficult) aspects to specify is the impact list, or the binding between variabilities and quality. Knowing such list is difficult even for the more experienced engineers, and we recognize that this is one of the most critical points for our approach. In order to mitigate this problem, we are investigating (at the time in which this thesis has been written) the idea of building several complete variants of a system, concretely measure/predict their quality, and infer from such data the impact lists similarly to the work described in [106].

- **Mismatch between quality properties and MDQP techniques**. The careful reader should have noted that all the quality properties we defined in this chapter for the ORM example refer to concepts of the input model, i.e., of the domain entity model. This has the advantage of showing quality information to the end-user at the same abstraction he/she is working. In detail, we specified for the response time and the wasted space metrics a domain context pointing to the domain entity with respect to which they are computed. However, MDQP techniques in general work at lower abstraction levels, where enough information is available to compute predictions. For example, in our case the analysis technique we use works at the abstraction level of the database schema model. As a consequence, it is possible to create a mismatch between the numbers computed by analysis techniques, the quality metrics we define, and the predictions we show to the end-user. We did not specify this in detail in this chapter, but resolving this mismatch is a matter of the MDQP techniques which, in order to inter-operate with QVT-Rational must also be able to translate back results from lower abstraction levels to higher ones.

5. Supporting The Designer

In this chapter we describe how QVT-Rational concretely generates feedback for system designers. Here we concentrate on the runtime aspects, that is, how quality-driven model transformations are actually executed, and we describe to which degree we automate the tasks usually performed manually by designers. This chapter is structured as follows.

First we give a brief recap on the role of the designer and the tasks he is responsible for when using QVT-Rational. We describe how requirements about non-functional properties of a system can be specified by using our requirements language, and we demonstrate this in the context of the ORM example.

Then we describe how quality-driven model transformations are actually executed. First we describe the abstract execution semantics, by showing how the execution of quality-driven model transformations corresponds basically to exploring the tree of all the possible system variants that may be generated by the transformation itself. Then we dig into the runtime of QVT-Rational, we show how High-Order Transformations (HOTs) are used to support concrete execution and we describe in detail all the steps necessary for the exploration of the variants.

Finally we describe how requirements can be used at runtime to improve automatic exploration of the design space, by guiding the traversal toward the most promising areas of the design space.

All the argumentation in this chapter is reified by means of the ORM example we described in the previous chapters.

5.1. Recap on the Designer Role

In the previous chapters we used Figure 5.1 to outline the QVT-Rational approach, and we concentrated specifically on the upper part of the figure, that is, the tasks that must be performed by domain experts to program QVT-Rational for a particular application domain.

In this chapter we instead concentrate on the lower part of the figure, that is, all the artifacts that are produced by and with which the domain expert interacts. In detail, from the perspective of the system designer, interacting with QVT-Rational requires:

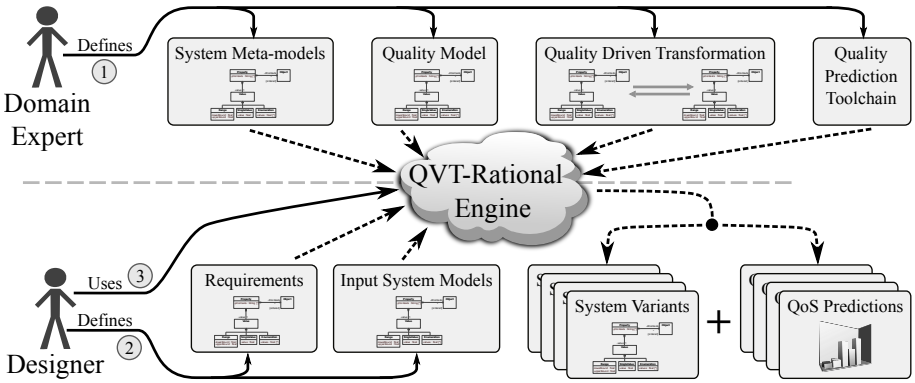

Figure 5.1.: Overview of QVT-Rational.

- **Defining the Input Models.** Obtaining feedback with QVT-Rational requires executing the model transformation defined by the domain expert which embeds information about how to generate system variants with different quality properties. However, executing the transformation in turn requires the availability of the input system models to feed the transformation. The system designer is responsible for creating such artifacts, which represent the system being designed or some of its facets. If we consider the ORM example, this step would correspond to generating the class diagram corresponding to the domain entities of interest for the application being developed. It is indeed worth noticing here that the system designer is not the only source of the system input models. Generative techniques can be used to automatically derive (at least partially) some of them. System model may be even generated by QVT-Rational, in the rather common case in which our framework is used at several abstraction levels. Designing systems is however a complex task outside the scope of this thesis. Here we concentrate on the steps which come after, i.e., an initial system design is available, the exhibited quality does not meet the designer's requirements, and QVT-Rational is used to generate system variants which respect the desired QoS. As a matter of fact, in the rest of this chapter we do not further detail this step but we assume that system models have been already generated for us.

- **Defining Requirements.** We deal with quality of system designs and, specifically, QVT-Rational aims at generating the system variants which exhibit the required QoS level. The human party in charge of specifying what is an acceptable QoS level (or

the non-functional requirements for the system) is the system designer. Without a specification of such concern, it would be impossible from our side to identify which of the viable system variants are valid (with respect to the required QoS level) or not. In the case of our ORM running example, defining an acceptable QoS level would correspond to requiring that the response time of all queries does not overcome a certain threshold, or that the wasted space is the lowest possible.

- **Obtaining Feedback.** The last step which may be performed by the system designer is feeding our system with the necessary input system models, with the non-functional requirements, and obtaining feedback about quality in the form of the viable system variants exhibiting a satisfactory QoS level. This process, as we will see in this chapter, may be either fully automatic or partially manual. In the former case, the process of exploring the design space and of generating viable system variants is entirely handled by our framework, in the latter case the system designer is involved in the exploration and actually guides the exploration by specifying which alternatives should be picked for each variability. It is worth noticing here that the viable output models are not the only artifacts produced during the feedback generation process, but they are accompanied by the quality predictions computed during the process. We also recall that requesting feedback to QVT-Rational is not a *single-shot* operation. Feedback is a process well described as a loop, the system designer may require feedback several times, for different system concerns (possibly at different abstraction levels), at different times and for different design models.

5.2. Defining Requirements

We provide the language depicted in Listing 5.1 to define requirements about non-functional properties. The advantages of providing a formal meta-model (and the associated textual notation depicted in the listing) to define non-functional requirements are two-fold. First, it serves the purpose of making more formal a design concern which is usually only a matter of the documentation and addressed in an informal way. In most cases, natural language and textual documents are used to express requirements, which makes impractical a structured management of requirements across the whole development cycle and their usage in development tools. We work in an MDSD setting; as a matter of fact, the most appropriate way of representing non-functional requirements

is through models and DSLs. Second, using a formal language to express requirements opens the doors to their usage in development tools and, most important, to their usage in our feedback process. As we will see next, requirements expressed with our language can be checked against system models accompanied with quality predictions, and helps QVT-Rational in further automating the process of finding good system variants in the design space.

Listing 5.1: The grammar of the requirements language.

```
<reqs_model> ::= <import_def>* <reqs_list>
<import_def> ::= 'import' <id> ':' string ';'
<reqs_list>  ::= 'requirements' <id> ':' (req)+
<req> ::= ('soft')? 'req' <id> '{' <bool_expr> '}' ';'

<bool_expr> ::= <primary_expr> <bool_op> <primary_expr>
<bool_op>   ::=  'and' | 'or' | 'xor' | '=>'
<primary_expr> ::= <bool_literal> | <not_expr> |
                   <req_ref_expr> | <comp_expr> |
                   ('(' <bool_expr> ')')

<bool_literal> ::= 'true' | 'false'
<not_expr>     ::= 'not' <primary_expr>
<req_ref_expr> ::= '@' <id>
<comp_expr>    ::= <belongs_expr> | <number_comp_expr>

<number_comp_expr> ::= <prop_expr>
                       <comp_op> ('[' <number> ']')?
                       <number>
<comp_op> ::= '==' | '<=' | '>=' | '<' | '>' | ''

<belongs_expr> ::= <prop_expr>
                   'in' ('[' <number> ']')
                   <set_expr>
<set_expr> ::= ('(' <number> ',' <number>')') |
               ('{' <number> (',' <number>)*'}')

<prop_expr> ::= '$' <id> ('!' | '!!')?
                ('(' <param> (',' <param>)* ')')?
                ('[' <index> ']')?
<param> ::= '_' | <string> | ('#' <id>)
<index> ::= 'max' | 'min' | 'var' | 'avg'
```

Non-functional requirements expressed through our language are boolean expressions over the quality metrics previously defined by the domain expert with the *NFPs* meta-model. The *import* statement serves exactly to this purpose, i.e., importing into a requirements model the quality metrics over which requirements will be defined. Requirements can be

specified by combining through standard boolean operators (and, or, xor, not, implies) comparison expressions over quality metrics or expressions referencing other requirements. This option is useful to define complex logical requirements which may be considered a combination of more simple requirements, it can be specified via the @ notation, and it is a convenient mechanism to favor reuse and modularization.

Comparison expressions are instead the most important and complicated construct of the language we propose. As the grammar in Listing 5.1 shows, comparison expressions may be of two kinds; they may be set comparison expressions or number comparison expressions. What distinguishes these two kinds of expressions is the type of the Right Hand Side (RHS). In the former case it is a set (either an enumeration of all the values with the brackets notation may be specified, or upper and lower bounds may be specified with rounded brackets) and the only operator which may be applied is inclusion (*in*). In the latter case the type of the RHS is a number and any of the usual comparison operators (e.g., equals, less or equal, greater or equal) may be used. Comparison may be evaluated exactly or a delta (by appending it with square brackets after the operator) can be specified in order to tolerate possible prediction errors and floating-point computation imprecisions.

In both cases (a set or a number for the RHS), the LHS of a comparison expression must be a property reference expression (indicated with the $ notation). Property reference expressions are similar to selections and projections available in the SQL language to query relational databases. They specify where the values to evaluate a comparison can be retrieved, by stating from which quality metrics predictions must be selected, for which contexts (with the same meaning we specified for *NFPs* metamodel in Section 4.4) they must be selected, and what kind of results must be retrieved (a single value, the whole data-set computed by the MDQP analyzer, or a statistic over the data-set such as the average, the minimum, or the maximum). To specify values for the contexts, the same syntax we use to bind quality metrics to variabilities can be used: values for primitive contexts may be specified with a string, values for domain contexts can be specified by referencing a model element, and in both cases the catch-all operator can be specified to match all the possible values compliant with the context. Much more details will be provided in a short while talking about the evaluation semantics, in the immediate we will clarify how requirements can be expressed with the help of the ORM running example.

5.2.1. The Requirements for the ORM Running Example

We recap that in the ORM running example we care about two quality concerns (i.e., response time of queries and wasted space) and that we defined three quality metrics (i.e., the global response time, the local response time, and the wasted space). Listing 5.2 shows the requirements we define for it.

Listing 5.2: The requirements for the ORM running example.

```
1  import NfpsModel : "Uml2Rdbms.nfps";
2
3  requirements OrmRequirements:
4    req GlobalRtIsOk {
5      $GlobalResponseTime(_)[avg] < 30
6    }
7
8    req LocalRtContributionIsOk {
9      $ResponseTime(_,_)[max] in (20,40) and
10     $ResponseTime(_,_)[min] > 5
11   }
12
13   req RtIsOk {
14     @LocalRtContributionIsOk and @GlobalRtIsOk
15   }
16
17   req WsIsOk {
18         $WastedSpace(_) <=[0.01] 0.2
19   }
```

The code is as simple as self-explanatory. On Line 1 we use an import statement to declare that the quality metrics defined in the *Uml2Rdbms* quality model should be made available for usage in requirements definitions. Lines 3-6 define the *GlobalRtIsOk* requirement, which imposes that valid solutions should exhibit an average *GlogalResponseTime* value for all the queries defined in the usage profile (we use the catch-all operator to specify this) not exceeding the 30 ms threshold. Lines 8-11 declare the *LocalRtContributionIsOk* requirement, which instead references the local response time property. It states that the maximum contribution of each domain entity to each query (again we use the catch-all operator) should be in the $(20, 40)$ ms interval and that the minimum should be greater than 5 ms. This clause (the last one referring to the minimum) is an extra condition we impose in order to not over-optimize the database system. Lines 13-15 define instead the *RtIsOk* logical requirement, which is the composition of the previous two requirements about the response time of queries. Basically, its value will be equal to the conjunction of

the values of the other two requirements. The last requirement defined on Lines 17-19 (*WsIsOk*) concerns wasted space, and imposes that such ratio must not be greater than 0.2 with an error of 1%.

The careful reader should have noted that in the text we specified the measure (i.e., milliseconds [ms]) of the thresholds for the response time metric, but such piece of information is missing in our requirements. This is not an error, but a thought choice we took while designing the requirements meta-model and its associated textual DSL. The kind of a quality metric (if it is a scalar or not) and its scale can be specified in the quality model. This information is used for example by MDQP analyzers to produce values for predictions in the right format, and is used also by our requirements evaluation engine to correctly interpret the numeric values used in the requirements specification. This is the reason why scales and measures may be safely omitted inside requirements definitions.

5.2.2. Evaluation Semantics

So far we have briefly described the constructs we provide to define requirements and we showed an example of their definition in the context of the ORM example. What we must still clarify is how requirements are actually evaluated, or their evaluation semantics. In this sense, the most important (and complicated) aspect that needs to be clarified is how property references and comparison expressions are evaluated. The evaluation of the boolean expressions built upon such clauses is well-known and we believe it does not need any further clarification.

In order to ease the comprehension of the evaluation semantics we will proceed by example. In detail, let us consider again the reference ORM example and suppose that we are translating the domain entities depicted in Figure 5.2. Suppose also that in the usage profile of the application we define two queries accessing the domain entities, the *FindPerson* query which seeks for a certain person given its name and the *CheckOut* query which seeks for a given cart, examines all the items placed inside it and computes the total amount to be paid. Finally, let us consider one of the possible database schemas corresponding to the depicted entity model, and suppose that the results of the predictions for the response time of the considered queries are the values listed in Table 5.1. All the values are computed in milliseconds and the table cells where the not-available (n.a.) symbol has been placed correspond to those values which could not be computed by the MDQP analyzer. There are in fact situations (usually much more complicated than this) in which simulations and analysis tools fail to compute a prediction, for

example because not enough statistical confidence was reached. For the sake of simplicity, let us suppose that this was the case for the *Item* entity and the *CheckOut* query.

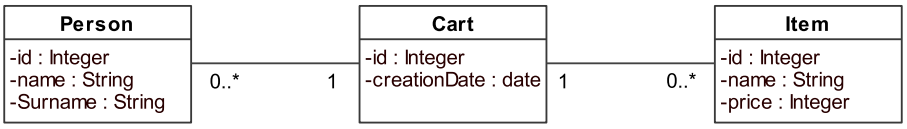

Figure 5.2.: A small entity model.

Query	Entity	Response Time [ms]			
		Values	Min	Max	Avg
FindPerson	Person	16, 19, 22, 19	16	23	19
CheckOut	Cart	15, 21, 24, 19	15	24	19.75
	Item	n.a	n.a	n.a	n.a.

Table 5.1.: The predictions for the response time.

In such a setting, the evaluation of the *LocalRtContributionIsOk* requirements would yield a successful result. This requirement in fact selects all the available values in the *Max* and in the *Min* column, which in turn satisfy the set inclusion constraint and the greater constraint, respectively. If not differently specified, when a not-available value is selected it is ignored in the computation of a comparison expressions. Other behaviors are possible, i.e., the *strict* and *super-strict* modes which we will describe in a short.

In the example we just showed that the property selection expression in the comparison yields a list of numbers. What we see now is how evaluation works when instead a list of lists of values is selected. This is the case if we consider the following new requirement

```
req RequirementExample {
  $ResponseTime(_,_) in (5,40) and
}
```

What happens in this case is that, instead of selecting a statistic for the response time metric, we select all the values provided by the MDQP analyzer, i.e., the ones listed in the third column of Table 5.1. The values yielded by the property selection expressions are not single values but sets of values and they are treated consequently in the evaluation. Both the set inclusion operator and the standard numeric operators can be used in this situation. In the former case standard inclusion with the RHS set is checked; in the latter case, each value of each set is compared to the RHS number.

Strict and Super-strict Evaluation

We mentioned previously that we provide two different behaviors to handle missing values. It is in fact frequent when analyses are complex that predictions for all the required metrics cannot be computed. For example, if a time limit is specified to interrupt long running QN simulations, it may be possible that the results computed by the MDQP tool are incomplete. Another situation, still concerning QN, happens when the system does not reach a steady state (either because there is no steady state or because the simulation time required to reach it is too long) and the MDQP analysis tool is not able to compute predictions with the required statistical confidence.

In some cases, this kind of situations deserve a special handling. The default behavior in case of a missing value is to silently ignore this during the evaluation. This is the behavior we showed before. However, in some cases it is necessary to gracefully signal this fact to the designer. By doing this, the designer can be notified that he/she cannot fully trust the quality of the solution identified by our system, and also our automatic exploration engine can take advantage of this information.

In detail, we provide two different strategies to deal with such situations: the strict strategy (imposed by placing a *!* in the property selection expression) and the super-strict strategy (imposed by placing *!!*). The difference between these two modes is rather subtle. In the former case, if a missing value is found the event is notified and the evaluation of the requirement proceeds but yields an unsuccessful value (i.e., the requirement evaluates to false). In the latter case, the evaluation fails and the entire process stops.

Soft and Hard Requirements

The careful reader should have noted in the grammar depicted in Listing 5.1 that the keyword *soft* can be prepended to a requirement definition. To handle trade-offs between non-functional properties we provide the ability to define two different types of requirements: soft requirements and hard requirements. Hard requirements are the default (i.e., if no *soft* keyword is prepended the requirement is considered hard) and are required to hold in order to consider a design alternative valid. On the contrary, soft requirements must be manually specified and are not required to hold to consider a design solution valid. Soft requirements are a convenient mechanism to express preferences between valid solutions and non-functional properties. A solution satisfying soft requirements in addition to hard requirements is considered by QVT-Rational a better

solution than one satisfying only hard requirements. In practice, soft requirements provide a mechanism to organize solutions according to the importance of quality properties and serve the purpose of declaring to QVT-Rational situations in which competing quality properties come into play.

Let us consider the ORM example and the requirements we specified in Listing 5.2. Performance of queries and wasted space are two competing quality properties. If we consider the hierarchy mapping variability, using an upward collapse strategy improves performance of read only queries (a least number of joins is required to retrieve all the data for each record) at the extra cost of wasting storage space. If for a designer performance of queries deserves much more attention than wasted space, an option to signal this trade-off to QVT-Rational is to declare the *WsIsOk* as a soft constraint

```
soft req WsIsOk {
      $WastedSpace(_) <=[0.01] 0.2
}
```

This change has the following effect on QVT-Rational while exploring the design space: the priority will go to the optimization of query performance, wasted space will be optionally considered, and two solutions exhibiting the same query performance will be ordered according to the wasted space metric (the lower, the better).

5.3. Executing QVT-Rational Transformations

Generating feedback in the form of the viable design alternatives (accompanied by quality predictions) satisfying the quality constraints imposed by the designer requires the ability to execute the quality-driven model transformation we defined in the previous chapter. Executing such transformations is however not as simple as executing a standard model transformation. First of all the outcome of the transformation is not single, the transformation embeds variability definitions, it is on purpose non-injective, and several viable outputs may be potentially generated. Second, while exploring all the possible variabilities, variants must be evaluated, their quality predicted, and requirements must be checked in order to select only solutions valid with respect to the willings of the designer. Third, the standard execution model of the underlying model transformation languages (QVT-Operational and QVT-Relations) we extended does not contemplate such kind of execution model, i.e., where multiple outputs are viable, where variants exploration must be controlled, and where extra steps (such as quality prediction and requirements checking)

must be executed.

From an abstract point of view, executing a QVT-Rational quality-driven model transformation corresponds to exploring the design space, to identifying possibly all the viable design alternatives, to evaluating them, and to checking if they are valid with respect to requirements. This general framework is well represented by the loop depicted in Figure 5.3. Given a quality-driven model transformation where variabilities

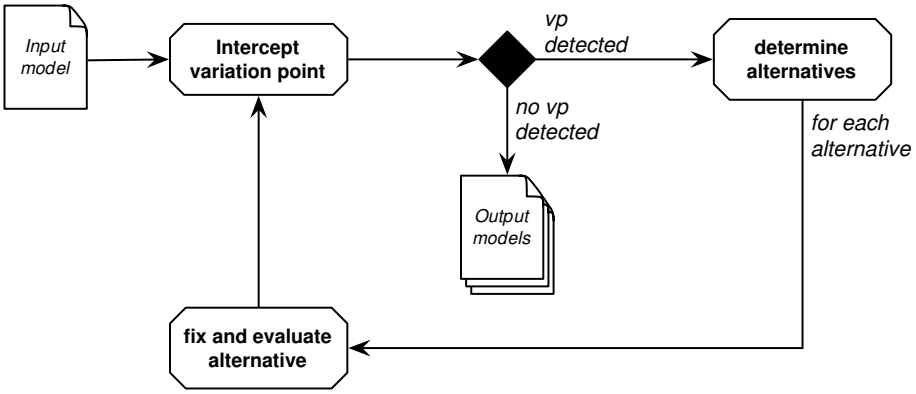

Figure 5.3.: The general execution loop.

have been defined and the input models for the transformation, we first intercept the occurrence of variation points. Given a variation point definition and the transformation instructions used to express it, we say that it occurs when we reach those instructions during the execution of the transformation, and none of the viable alternatives for that variation point and for the modeling objects involved by those instructions has been already selected. For example, let us consider the partitioning variability we defined in the ORM transformation and the input model shown in Figure 5.2. For each domain entity (i.e., *Person*, *Cart*, *Item*), we encounter the partitioning variability (i.e., we execute the *UsePartitioning_VariationPoint* operational mapping in case of the QVT-O based transformation, or for QVT-R the predicates in the where clause of the *ClassToTable* relation expressing the variation point) three times, one for each domain entity. When a variation point is intercepted we explore all the possible variants (i.e., using either a single table or a partition of tables), we evaluate the quality of all the candidates, and we iterate again over the loop. If no variation point is intercepted — all the viable variabilities have been explored and we are set with exploring the design space — the transformation ends and all the output models

produced constitute the output of the transformation.

In practice, this execution schema is akin to exploring the tree of all the possible design variants. It can be thus represented by a tree where inner nodes represent the occurrence of variation points and where leaf nodes represent either valid or invalid solutions. We will use this kind of representation several times in the following, and an example to clarify the concept is depicted in Figure 5.4. The figure outlines an hypothetical branch of the exploration tree, i.e., the branch depicting the occurrence of the three variabilities we defined in the ORM example for a single domain entity. Each occurrence of a variation point (a diamond in the figure) leads to the generation of two variants, until no more variation points occur and the final models (leaves *o1*, *o2*, *o3*, and *o4*) are produced. The decisions taken from the root to each leaf list the decisions taken during the process. For example, for the output model *o1* the upward collapse strategy, the randomized generation of identifiers and not using partitioning are the variants selected to resolve all variabilities.

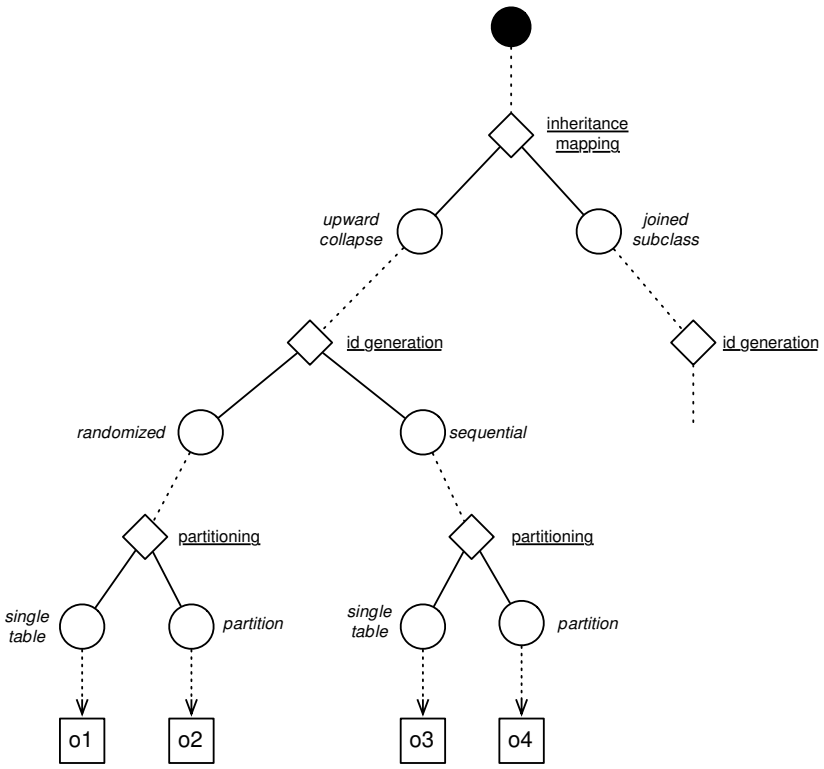

Figure 5.4.: An hypothetical execution tree.

5.3.1. Digging into the Details

We just presented a simplified view of the execution process. Executing our quality-driven model transformations is a much more complicated process involving several analysis, preparation, and sub-transformation stages. A much more detailed picture of the execution cycle is represented in Figure 5.5: dashed ovals represent steps in the cycle, diamonds represent branches, solid arrows stand for control flows, and dashed lines connect steps with produced artifacts.

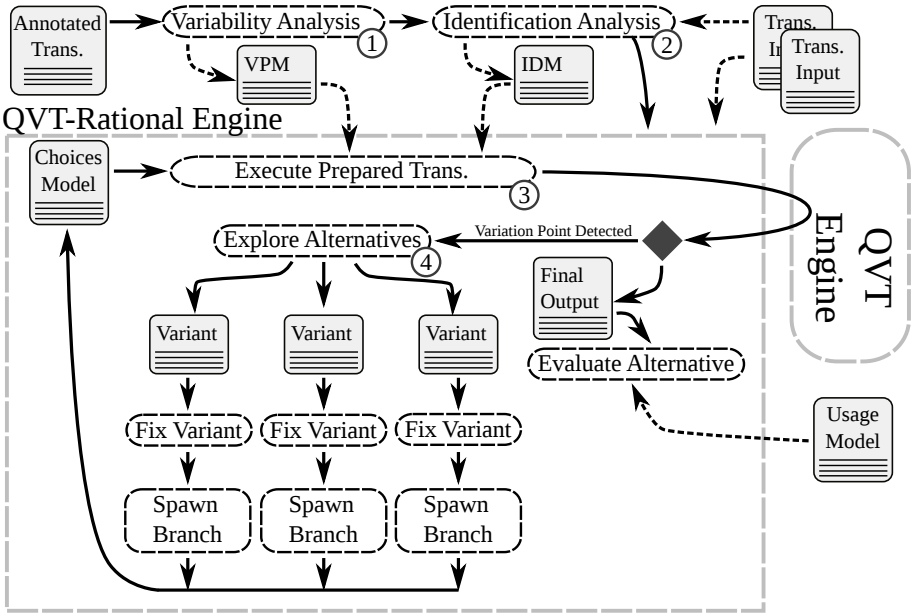

Figure 5.5.: The QVT-Rational Execution Cycle.

The general idea is rather simple. To generate all the possible system variants the quality-driven model transformation is executed several times, and concrete execution is not performed by us but is delegated to existing interpreters. We come into play only when a variation point occurs and control is transfered from QVT interpreters to our engine. When this happens, the QVT-Rational engine identifies the possible candidates, copies all the transformation artifacts (e.g., the input/output models, the QVT-Rational models), saves the information about the specific alternative (a step which we refer to with the term *fixing*), and spawns for each alternative a new exploration branch. For each new exploration branch iteration over the loop continues until all no new variation point is intercepted, a final solution is identified, its quality evaluated, and requirements checked. In the following we describe in

detail all the important aspects of this process.

Three are the inputs for the global process: the *Annotated Transformation* defining the feedback rules, the *Input Models* for the transformation, and the *Usage Model* specifying under which conditions the non-functional characteristics of the system should be evaluated. To clarify what these models are, let us consider again the ORM case study. The *Annotated Transformation* is the transformation created by the domain expert which contains the information about design alternatives, the *Input Models* correspond to the class model (using the UML terminology) which represents the domain entities, while the *Usage Model* contains information about the queries that will be performed over the domain entities.

5.3.2. Analysis Stages

Before a QVT-Rational transformation can be actually executed it must undergo two analysis stages: the *variability analysis* and the *identification analysis*.

The *Variability Analysis* parses the transformation source code, retrieves the Abstract Syntax Tree (AST) [100] of transformation which will be used to manipulate the transformation and prepare it for execution, and searches for variability annotations. The outcome is a *Variation Point Model* (VPM) — based on the *Variability* meta-model we previously defined in Section 4.3.1 — capturing all the necessary information about defined variation points, variants, their properties (i.e., impact on non-functional properties and how to evaluate them), all linked to the AST model concepts which point to the transformation source code constructs expressing those variabilities.

The *Identification Analysis* instead traverses the transformation AST in search of *key* definitions, which specify how to identify objects in a transformation. In case of Relational transformation it searches for the language native constructs, while in case of Operational transformations it searches for the *@keydef* annotation serving the same purpose. As we announced while describing the annotation language, this information is necessary during execution to link variation points and design decisions to the modeling objects involved in the transformation. To explore the design space and generate feedback we execute several quality-driven model transformations, and the same model elements must be uniquely identified and equally treated during all these executions. Using the same object pointers used by the QVT execution engines is not feasible, since they change across different executions. In order to overcome this problem, we extract the key definitions by ourselves and we use them to

manually build the corresponding model elements identifiers.

Once *key* definitions have been parsed, this analysis traverses all the objects defined into the transformation *Input Models*, builds their concrete identifiers, and saves all this information inside the *IDentification Model* (IDM). In order to clarify this point, let us consider for example the key definitions we specified in Listing 4.13 which we report here for the sake of clarity and the domain entity model depicted in Figure 5.2. Table 5.2 shows some of the identifiers built by the identification anal-

Listing 5.3: Defining meta-class keys with QVT-O.

```
1  @keydefs{
2   simpleUml::Package(#name),
3   simpleUml::Classifier(#name, #namespace),
4   simpleUml::Class(#name, #namespace),
5   simpleUml::Attribute(#name, #owner),
6   simpleUml::Association(#name, #owner),
7   simpleUml::Generalization(#name, #owner),
8
9   simpleDb::Schema(#name),
10  simpleDb::Partition(#name, #schema),
11  simpleDb::Table(#name, #schema),
12  simpleDb::Column(#name, #owner),
13  simpleDb::Key(#name, #owner),
14  simpleDb::ForeignKey(#name, #owner)
15 }
```

ysis. In particular, the table shows the identifiers for the root *Package* enclosing the domain entities and for the *Person* domain entity. Cross-references are expressed with the # notation when references are used to identify a meta-class. Identifiers for the model elements not shown in tables can defined in the same way, we omit them for the sake of clarity.

The careful reader should have noted that we define keys both for the input and for the output models, but we said that the identification analysis traverses only the input model and builds concrete identifiers only for the model elements defined in there. However, during the execution cycle we need to identify also the elements in the output model, but such elements are not available until they are created during the transformation. For example, we need to identify also the *Schema*s, the *Table*s, and the *Column*s created during the execution of the ORM transformation. We save the information about key definitions in the IDM for this purpose, i.e., in order to defer at runtime the update of the IDM with the identifiers for output model elements.

Model Element	Type	Id	Keys
SimpleModel	Package	1	name=‘SimpleModel’
Person	Class	2	name=‘Person’, namespace=#1
id	Attribute	3	name=‘id’, owner=#2
id	Attribute	3	name=‘id’, owner=#2
name	Attribute	4	name=‘name’, owner=#2
surname	Attribute	5	name=‘surname’, owner=#2

Table 5.2.: The identifiers for the small entity model.

5.3.3. Preparing the Transformation

A quality-driven model transformation, specified with our extension of either QVT-O or QVT-R, must be slightly modified and prepared for the execution. For example, in case of QVT-R we need to add new relations to intercept when variation points occur during the execution and their black-box implementation. The same holds for QVT-O based transformations, where to intercept variation points we need to add the constructs to invoke our black-box library. In the following we describe how we prepare the transformations in both the cases.

Preparing QVT-R Based Transformations

We leverage black-box relations to intercept when a variation point occurs. The general idea is to substitute each predicate defining a variation point with a call to a compatible black-box relation (a relation with the same signature of the variants for the variation point), which will be responsible of intercepting the execution of the instructions defining the variation point and notifying to our engine this fact. How we substitute the predicates expressing a variation point and which new instructions we put in place of them will be described in the next sections, here we concentrate on the black-box relations we add to the original transformation.

To prepare the model transformation for execution we perform an High-Order Transformation (HOT)[48], i.e., a transformation taking in input and producing as output another transformation. Concretely, the HOT works on the AST of the original quality-driven transformation

produced during the *Variability Analysis*. A nice consequence of using MDSD techniques in QVT-Rational is that everything is a model, even AST are models, can be extracted and represented through the same techniques available to manage design and development of systems, and specifying HOTs is consequently simple.

As we anticipated, the duty performed by our preparation HOT is to add for each variability a new black-box relation compatible with the signature of the variants. The HOT receives thus also the *VPM* produced during the *Variability Analysis* in input and works as follows: it searches all the variabilities defined in the *VPM*, it identifies the QVT-R constructs (i.e., the relations and the predicates) used to express them which are linked to the *VPM*, and adds a new compatible black-box relation for each variation point.

To clarify which kind of new relations are added to the original transformation we may refer again to our ORM example. Let us consider for example the code we used to defined the *UsePartitioning* variation point, which we reported here in Listing 5.4 for the sake of clarity. The signature of the variants accepts an object of type *Class* and an object of type *Table*. The compatible black-box relation (which we call *black-box interceptor*) we add to the transformation is depicted on lines 26-31 of the same listing, it accepts the same kind of objects and declares via the **implemented by** keyword the name of the black-box method defining its implementation.

What we have seen so far is only part of the process. We in fact added only the declaration of the black-box interceptor, but we have not detailed how we define its behavior. From an abstract point of view, the task of a black-box interceptor is to identify when a variation point occurs and to communicate to QVT-Rational this fact, i.e., to communicate the unique identifier of the variation point and the identifiers of the model elements participating to it. For example, if we consider again the simple entity model shown in Figure 5.2, we expect the *UsePartitioning_Interceptor* to be invoked three times during the execution of the quality-driven model transformation. One time for each domain entity defined in the input model. The events notified to QVT-Rational will be thus three, and will contain a reference to the *UsePartitioning* variability and to the *Person*, *Cart* and *Item* domain entities, respectively.

However, how black-box implementations are handled is still a matter specific to each QVT-R transformation engine. For example ModelMorf [102] — which we use for the implementation of QVT-Rational — handles them by using user-supplied Java implementations. QVT-Rational automates also this step, by using a JET-based model-to-text transformation [107] to generate the Java source code implementing the behavior

Listing 5.4: The partitioning variability in QVT-R and its black-box interceptor.

```
1  relation ClassToTable
2  {
3    domain uml c:Class {...}
4    domain rdbms t:Table {...}
5    where {
6      @varpoint{ name := UsePartitioning }
7      ClassToSingleTable(c,t)
8        xor ClassToTableInPartition(c,t);
9    }
10 }
11
12 @variant{ name := ClassToPartition }
13 relation ClassToTableInPartition
14 {
15   domain uml c:Class {...}
16   domain rdbms t:Table {...}
17 }
18
19 @variant{ name := SingleTable }
20 relation ClassToSingleTable
21 {
22   domain uml c:Class {...}
23   domain rdbms t:Table {...}
24 }
25
26 relation UsePartitioning_Interceptor
27 {
28   domain uml c:Class {...}
29   domain rdbms t:Table {...}
30   implemented by UsePartitioningBBox
31 }
```

of black-box interceptors. Without digging too much into the details, the black-box handler we generate inspects the identification model searching for the identifiers associated with the objects participating in the variation point. When found, the transformation is interrupted and such information is passed to the QVT-Rational to continue exploration.

Preparing QVT-O Based Transformations

The preparation process to which QVT-O based transformations are subject is similar to the one we just described for Relational transformations. For example, also in this case we leverage black-box mappings to intercept variation points. The differences reside obviously in the way

variability is expressed and changed by our preparation HOT, and in the way we declare and use black-box mappings and helpers to manage variability during execution.

The *VpInterceptorLib* black-box library is responsible to manage variability during the execution of a quality-driven model transformation. The functionalities it offers in the form of helper functions callable from a QVT-O transformation are two:

- **Intercepting a Variation Point**. This function declared in the black-box library is similar to the black-box interceptors we described for Relational transformations both for its syntax and semantics. The signature of the function using a Java-like syntax is

```
boolean interceptVariationPoint(
  List<Object> vpParams,
  String vpId)
```

  The *vpParams* argument contains the list of the arguments passed to the mapping which implements the variation point, while the *vpId* argument contains the unique identifier of the variation point. When this function is invoked the *IDentification Model* is inspected and identifiers for the mapping arguments are retrieved. If no identifier is already present in the *IDM* (for example because some of the arguments of the mapping point to newly created elements belonging to the output models of the transformation) a new one is created and the *IDM* updated consequently. If everything goes well and no errors are encountered the function returns true, otherwise false is returned to signal an error condition.

- **Deciding about Variants**. This function helps in deciding whether a particular variant should be picked or not when a variation point for which a decision has been already taken occurs. Its signature using a Java-like syntax is

```
boolean decideForVariationPoint(
  List<Object> vpParams,
  String vpId,
  String variantId)
```

  The *vpParam* argument contains the list of the the arguments passed to the mapping which implements the variant, the *vpId* argument contains the unique identifier of the variation point to which the variant belongs, while the *variantId* argument contains the unique identifier of the variant for which this helper function has to decide. As it will be clear in a short, this function is used in

the *when* clause of mapping operations implementing variants to check if a particular variant should be picked or not for a specific occurrence of a variation point. In practice, the function inspects the list of decisions already taken during the exploration of the design space, and if a decision matches the particular variation point, variant, and list of arguments the function returns true.

The HOT responsible for preparing the quality-driven model transformation for execution is responsible for performing the changes outlined in Listing 5.5. First the black-box library providing the necessary helper functions to manage variability at runtime is imported in the transformation by adding the **import** statement shown on line 1. Then the following modifications are applied for each operational mapping used to define either a variation point or a variant.

Mappings responsible for defining variation points are modified by adding a new variant at the end of the disjunction (or merging clause in case of optional variation points). For example, in the case of the *UsePartitioning* variability we add the *UsePartitioning_Interceptor* mapping on Line 8. This new alternative for the variation point acts as a fall-back; when no alternative has been already selected the fall-back mapping is invoked and the variation point is correctly intercepted. The preparation HOT also adds the declaration of the fall-back mapping to the transformation AST, shown on lines 10-23 of the listing. For each variation point a specific interceptor mapping is defined, and the code to invoke the *interceptVariationPoint* function defined in the black-box library is generated. For example, in our case the HOT generates the code to create an ordered list containing only one object, i.e., the *Class* passed as the mapping context.

As we anticipated, also the operational mappings implementing variants are modified by the preparation HOT. In the specific case, the HOT adds (or modifies if one already exists) a *when* clause invoking the *decideForVariationPoint* helper function. Lines 27-34 in the listing show the *when* clause added by the HOT to the mapping implementing the *ClassToSingleTable* variant. The invoked helper function returns true if and only if a decision about the specific variation point, variant, and modeling objects already exist. In this case, the *when* condition consequently evaluates to true and the mapping is executed.

Listing 5.5: The partitioning variability in QVT-O and its black-box interceptor.

```
1  import it.polimi.qvtr2.operational.VpInterceptorLib;
2
3  @varpoint { name := UsePartitioning, ... }
4  mapping Class::UsePartitioning_VariationPoint()
5    disjuncts
6      Class::ClassToSingleTable_Variant,
7      Class::ClassToPartition_Variant,
8      Class::UsePartitioning_Interceptor;
9
10 mapping Class::UsePartitioning_Interceptor()
11 {
12         var res := interceptVariationPoint(
13                 Sequence{self.oclAsType(EObject)},
14                 "UsePartitioning");
15
16         if (res = null) then { // Interception was correct
17                 assert fatal (false) with log
18                   ("MAGIC MESSAGE ---> forcing termination");
19         } else { // Interception faulted
20                 assert fatal (false) with log
21                   ("MAGIC MESSAGE ---> Interception Failure");
22         } endif;
23 }
24
25 @variant { name := ClassToSingleTable, ... }
26 mapping Class::ClassToSingleTable_Variant()
27 when {
28         decideForVariationPoint(
29                 Sequence{
30                         self.oclAsType(EObject)
31                 },
32                 'UsePartitioning',
33                 'ClassToSingleTable');
34 }
35 {
36   -- The mapping body
37   ...
38 }
```

5.3.4. Intercepting and Generating Variants

Intercepting and generating the possible system variants is rather simple once the quality-driven transformation has been prepared for the execution. This is especially true for Operational transformations, while for QVT-Relations-based transformations the process is a bit more complex.

For an operational transformation, intercepting the occurrence of a variation point is just a matter of concretely executing the transformation. If no variation points occur it means that either no variability was present for the input models as specified in the model transformation, or that all variabilities have been completely explored. When this situation happens, the output model produced by the transformation is a final system variant, we may thus proceed with quality prediction by invoking the MDQP analyzers specified in the annotations and with the evaluation of the non-functional requirements. If evaluation succeeds, the model is considered valid and is saved for presentation to the designer, otherwise it is discarded.

When a variation point is intercepted (i.e., the fall-back interceptor mapping is executed and the helper function completes its execution without any problem) control is passed to the QVT-Rational engine. At this point, information about the possible variants is extracted from the *Variability Model* (VPM) and the exploration process forks. Forking means that for each possible variant all the artifacts (the model transformation input models and the QVT-Rational models) are copied, information about each specific variant associated to each forked process is saved in the *Choices Model* (a model which holds information about which variants are picked for each occurred variation point) by fixing the alternative, and exploration proceeds in parallel and independently for each fork.

In terms of the exploration tree, forking corresponds to creating a new branch in the tree. To explain the process let us consider again the simple entity model depicted in Figure 5.2. When the operational mapping corresponding to the *UsePartitioning* variation point is executed for the first time (let us suppose that it is executed for the *Person* domain entity) no decision has been already made, i.e., we have not explored the variation point. This is represented by node 1 in the exploration tree depicted in Figure 5.6. The *UsePartitioning* variation point exhibits two possible alternatives, using a single table for the mapping or using partitions. These alternatives correspond to nodes 2 and 3 in the exploration tree, and for each of them all the transformation artifacts are copied and a new independent exploration process starts. In the two forked processes the *UsePartitioning* variation point will occur again (nodes 4

and 5), but this time not for the *Person* domain entity (for which a decision has been already made and for which the *decideForVariationPoint* helper function in the *when* condition will evaluate to true) but for the *Item* entity. What happens now is identical, new branches will be forked (nodes 6-7 and 8-9) and the active exploration process will be four. The process continues in this manner, until no variation points occur and a final model is identified.

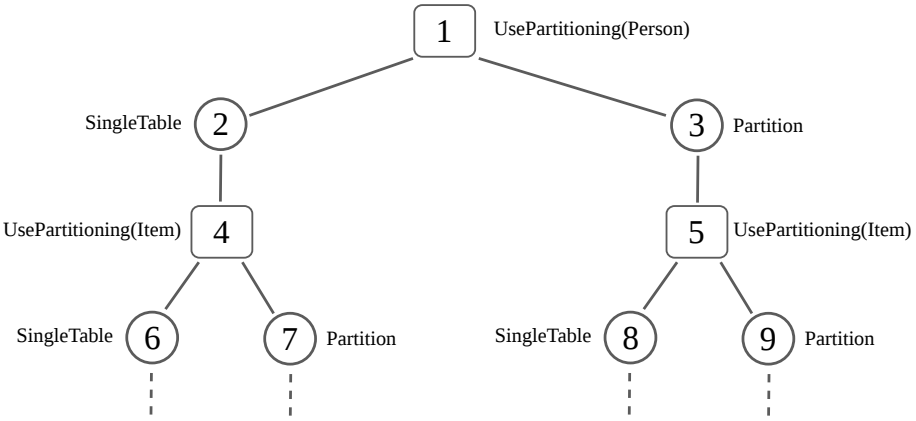

Figure 5.6.: A part of the exploration tree for the simple entity domain example.

QVT-R Based Transformations

In case of QVT-R the execution cycle is similar from a general point of view and for what concerns spawning new exploration branches, but identifying the occurrence of variation points and generating the variants is slightly more complicated. In the previous sections we showed how the preparation HOT changes the original quality driven transformation by adding black-box interceptor mappings and we outlined how we generate the code implementing those mappings. Here we clarify how these black-box interceptors are weaved in the transformation. Figure 5.7 highlights the differences in the execution cycle for QVT-Relations-based transformations. What changes is the presence of another step in the process, i.e., the building of a new *Intercepting Transformation* (IT) every time the loop starts. As it will be clear in a short, ITs are responsible both for the interception of variation points and of fixing variants when a decision about a variability has been already taken during exploration.

We recall that in a Relational quality-driven model transformation the variation points are expressed with a predicate in the *where* clause

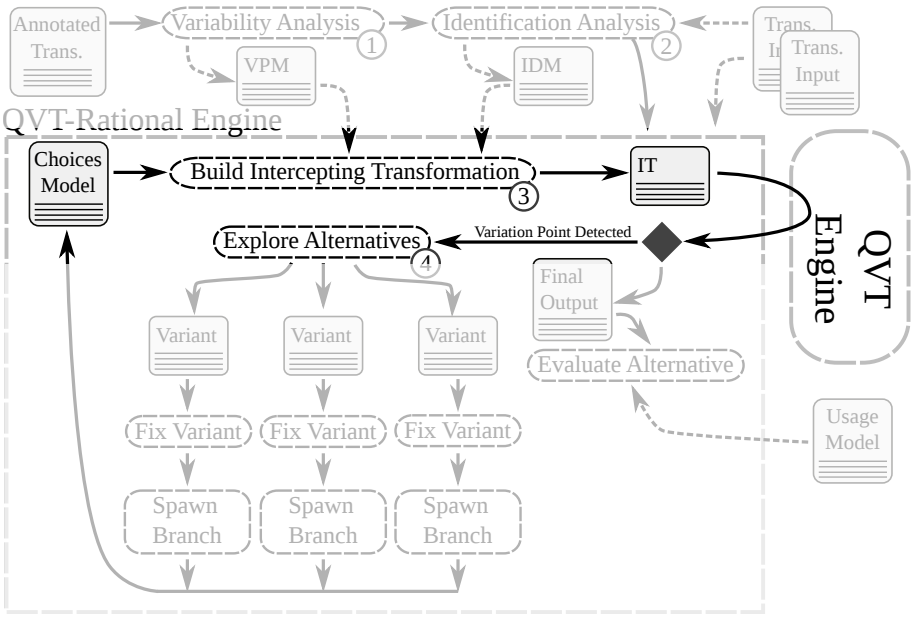

Figure 5.7.: Generation and execution of Intercepting Transformations.

of a relation listing all the possible variants. To intercept when such predicates are executed, the HOT responsible for creating ITs replaces the predicates expressing each variation point with a call to the black-box interceptor. Let us consider the *UsePartitioning* variation point shown in Listing 5.4. The mapping body is changed as shown in Listing 5.6 on line 6. The xor predicate disjuncting the viable variants is substituted with a relation call pointing to the interceptor corresponding to the variability.

Listing 5.6: Weaving black-box interceptors.

```
1 relation ClassToTable
2 {
3   domain uml c:Class {...}
4   domain rdbms t:Table {...}
5   where {
6     UsePartitioning_Interceptor(c,t);
7   }
8 }
```

Whenever such predicate is reached during execution, instead of invoking one of the possible relations implementing the variants the black-box interceptor is invoked. The model elements participating to the variation

point are thus identified, and control is passed to QVT-Rational in order to identify the possible variants and spawn new exploration branches. Note however that this is the most simple case, we must in fact intercept only the occurrence of variation points for which an alternative has not been selected. To discern between the two cases, the HOT creating Intercepting Transformations considers also the *Choice Model* (i.e., the model containing information about the already resolved variabilities and about previous decisions), and substitutes the transformation predicate representing the variation point with nested *if-then-else* constructs invoking selected alternatives for fixed variabilities, or invoking interceptors if this is not the case. Let us consider again the simple entity domain in Figure 5.2 and the corresponding exploration of Figure 5.6. When the variation point of node 1 occurs, two exploration branches will be spawned — node 2 corresponding to the *SingleTable* variant and node 3 corresponding to the *Partition* variant — and two Intercepting Transformations will be generated. In the case of node 2, we must not intercept the occurrence of the *UsePartitioning* variation point when *Person* is the entity class involved in the variability. The HOT responsible for creating the IT for node 2 replaces thus the predicate of the *UsePartitioning* variation point with the substitution predicate outlined in Listing 5.7 on lines 6-12. The

Listing 5.7: Weaving black-box interceptors with *if-then-else* constructs.

```
1  relation ClassToTable
2  {
3    domain uml c:Class {...}
4    domain rdbms t:Table {...}
5    where {
6      if (c.namespace.name = 'SimpleModel' and
7        c.name = 'Person')
8      then
9        ClassToSingleTable(c,t)
10     else
11       UsePartitioning_Interceptor(c,t);
12     endif
13   }
14 }
```

behavior of this snippet of code is rather simple. If for the entity involved in the variation point (which we identify by building the condition of the *if* construct with the information contained in the *Identification Model*) a decision already exists we invoke the fixed alternative (or we use the **true** predicate in case of a recursive variability and of the do-not-recur variant), in all other cases we invoke the black-box interceptor.

When decisions exist for different model elements, the *if-then-else* are nested in order to first check all the fixed variabilities and to fall-back at the end to the interceptor. Listing 5.8 gives an idea of this, by showing the IT for the branch corresponding to node 7 in Figure 5.6.

Listing 5.8: Weaving black-box interceptors with *if-then-else* constructs.

```
1  relation ClassToTable
2  {
3    domain uml c:Class {...}
4    domain rdbms t:Table {...}
5    where {
6      if (c.namespace.name = 'SimpleModel' and
7        c.name = 'Person')
8      then
9        ClassToSingleTable(c,t)
10     else
11       if (c.namespace.name = 'SimpleModel' and
12         c.name = 'Item')
13       then
14         ClassToTableInPartition(c,t)
15       else
16         UsePartitioning_Interceptor(c,t);
17       endif
18     endif
19   }
20 }
```

5.4. Exploration Modes

QVT-Rational provides two different execution modes to obtain feedback: an automatic mode where exploration of the design space is entirely automatic and managed by QVT-Rational, and an interactive mode where the designer is involved in the process and is responsible of selecting which variants should be picked and explored whenever a variation point occurs. In the following we describe both in detail.

5.4.1. Automatic Exploration

With automatic exploration the navigation of the design space is entirely managed by the QVT-Rational. The designer is involved in the process only at the end, when complete system variants have been already found, their quality evaluated, and requirements checked in order to identify only solutions compliant with the designer's willings.

When input models are large and several variabilities have been defined in the quality-driven model transformation responsible for generating feedback the design space may become quickly huge. If we also consider that MDQP analysis tools take usually long to performs their computations, the obvious consequence is that the feedback process may take an amount of time incompatible with the designer needs. The designer cannot wait hours to let QVT-Rational explore the whole design space. The direct consequence of this fact is that mechanisms to limit and guide the exploration must be provided in order to let QVT-Rational be useful. In the following we describe two of them which we implemented in our system: limitation of recursion and guidance.

Limiting Exploration

When recursive variation points come into play things may get complicated. We recall that a recursive variation point is a variation point where the relation/mapping implementing it recursively invokes itself. The nasty consequence is that the exploration tree, in the worst case, may be infinite. The variability we defined in the ORM example concerning how many slices are added into a partition is an example of this, and the exploration tree depicted in Figure 5.8 shows the situation. If we continue to pick the *AddAnotherSlice* variant, the depth of the tree increases indefinitely.

In order to overcome this problem, automatic exploration must be configured before its use by specifying the maximum recursion. In practice, this configuration parameter imposes a limit to the number of times a recursive variation point can be taken. Let us suppose that a limit of 2 has been imposed and let us refer again to Figure 5.8. With such a limit, only the first two *AnotherSlice* variants can be picked, the last one is automatically excluded by the limit and exploration terminates at the second-last node.

Another aspect concerning the limitation of recursion must be indeed tackled here. We recall that the same variation point may occur several times for different model elements. In the small example we used throughout this chapter the *UsePartitioning* variation point occurs three times, one for the *Person* entity, one for the *Cart* entity, and one for the *Item* entity. The same happens for the recursive *AddAnotherSlice* variation point. This aspect is important to understand how the recursion limit is applied during exploration. The recursion limit in fact is not applied globally, for example by counting the number of global occurrences of the *AddAnotherSlice* variation point. On the contrary, the limit is computed and applied locally, by counting the number of times the same

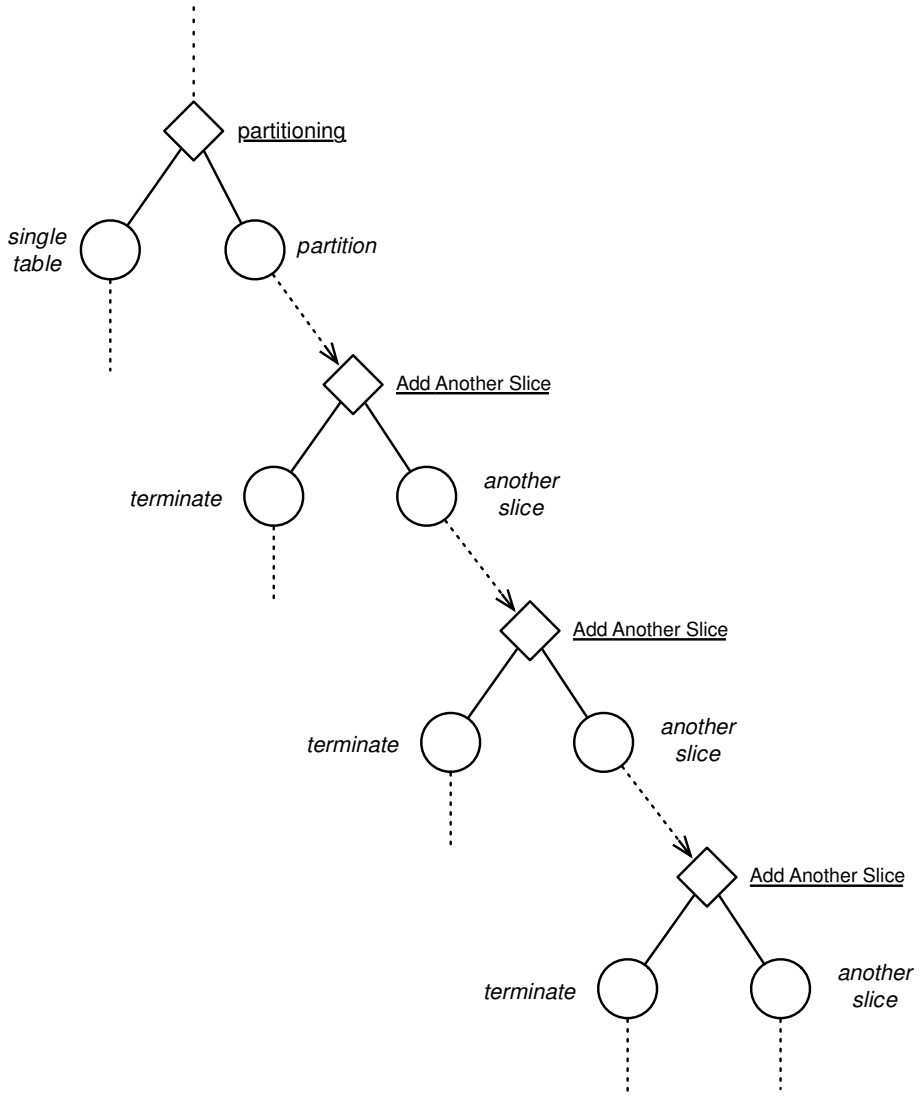

Figure 5.8.: An infinite exploration tree resulting from recursion.

variation point occurs for the same model entities. In the general case, all the contexts of a variation point (the arguments of a mapping for QVT-O, or the domains for QVT-R) are considered in the computation. This behavior is however not good in every case. For example, in our ORM transformation only the *Class* object passed to the mapping/relation must be taken into account, while the *Table* and the slice number should be excluded. The *subject* annotation comes into play for this purpose. It specifies which variation point contexts should be taken into

account while counting the number of local occurrences of a variation point.

In our ORM example we defined with the *subject* annotation that only the *Class* object should be taken into account for this counting. Without that specification, the recursion limit would not work since every time the *AddAnotherSlice* mapping is invoked with different arguments and the limit would never be reached.

Using Requirements for Guidance

When the design space is huge, in consequence of big input models and several variabilities, much of the time to execute the quality-driven model transformation may be spent in navigating areas of the exploration tree which are not promising. For example, let us consider our ORM example and let us suppose that partitioning for the *Person* entity is necessary to meet certain requirements about the response time of the database queries specified in the usage profile. The tree branch originating from the node corresponding to the *SingleTable* variant are useless, all the viable solutions in that sector of the tree will not be valid with respect to the requirements.

In order to overcome this kind of problems and speed-up the exploration process QVT-Rational provides a guidance mechanism which leverages requirements evaluation to identify the most promising areas of the design space. The guidance mechanism works by organizing the branches of the tree which must be explored with a Max-Heap [105], where on the top there are the most promising branches. The *'promise-value'* of a branch is computed by considering the value of the requirements after their evaluation.

At the beginning of the process, whenever a new variation point occurs (i.e., a new branch in the tree is created) the corresponding branch is put on the top of the heap. This is akin to exploring the design space tree with a depth-first policy, which turns out to be the fastest way of obtaining a first solution. We recall that complete system variants reside in fact at the lower frontier of the tree. Whenever a new complete solution is found, it undergoes quality prediction and requirements evaluation. If the evaluation succeeds, the solution is marked as valid and no guidance comes into play. When evaluation does not succeed some of the requirements where not satisfied. We inspect these failing requirements, we identify the most failing one, we identify the *interesting decisions* — the decisions taken to reach the invalid solution which had an impact on the failed requirements — and we schedule the visiting of the exploration branches corresponding to the *interesting decisions* with higher priority.

This is akin to preferring the exploration branches which may solve immediately a quality issue, by taking different strategies only for decisions which impact quality issues and by leaving unmodified everything else.

Once the most failing requirement is identified, finding the quality properties to which it refers and the decisions that must be changed is rather easy. Such information is in fact available in the *Variability Model* and in the *Choices Model*, and is derived from the rational annotations (in particular from the information defined in the *impact* annotation). Finding the most failing requirement is indeed a more complex task, which we carry out by computing the satisfaction level of each requirement. The less it is satisfied, the more failing the requirement is, the more its solving is critical.

We recall that requirements are, in the most complicated case, a boolean predicate over comparison expressions. To compute the satisfaction level of each requirement we first iterate over all the comparison expressions used in the requirement definition and we compute, for each one, its satisfaction level. For comparison expressions, the satisfaction level is computed by calculating the distance between the predicted value (i.e., the LHS of the expression) and the required value (i.e., the RHS of the expression). How this computation is performed depends on the particular comparison operator used and on the type of the LHS and RHS values, Table 5.3 summarizes how we perform such computation.

Operator	RHS Type	Result
$lhs = rhs$	Numeric	$euclideanDistance(lhs, rhs)$
$lhs < rhs$		
$lhs > rhs$		
$lhs <= rhs$		
$lhs >= rhs$		
lhs **in** rhs	Interval	$min\{$ $eclideanDistance(lhs, rhs.lowerBound)$, $eclideanDistance(lhs, rhs.upperBound)$ $\}$
lhs **in** rhs	Enum.	$min(\{$ $\forall v \in rhs.values \mid euclideanDistance(lhs, v)$ $\})$

Table 5.3.: Computing the satisfaction level of comparison expressions.

Once the satisfaction levels for each comparison expression are computed, they are aggregated in order to generate a single value for each requirement. Again, how aggregation is performed depends on the par-

ticular boolean operator used in the requirement definition, and Table 5.4 summarizes how we perform such computation.

Operator	Result
lhs **and** rhs	$min\{lhs, rhs\}$
lhs **or** rhs	$max\{lhs, rhs\}$
lhs **xor** rhs	$max\{lhs, rhs\}$
lhs **implies** rhs	lhs
not rhs	rhs

Table 5.4.: Aggregating satisfaction levels of comparison expressions.

Figure 5.9 clarifies the whole process by outlining what happens when guidance comes into play in the context of the ORM example. When solution corresponding to node 100 is found, the requirements concerning the queries accessing the *Person* domain entity fail and guidance comes into play. The decision taken at node 1 is about the *UsePartitioning* variability for the *Person* class, and it has an impact on that query referenced by the failing requirement. The branch spawning from node 2 in the exploration tree (which corresponds to changing the strategy to map the class with partitioning) is scheduled thus with high priority, and valid solutions will be likely to be found faster.

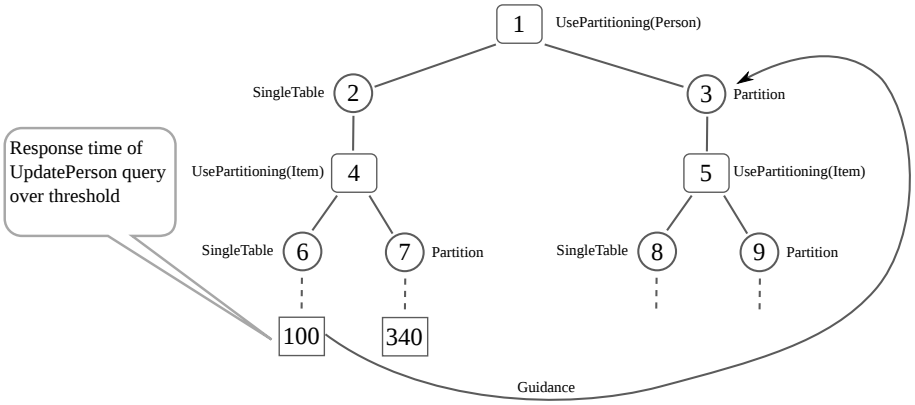

Figure 5.9.: Guidance while exploring.

5.4.2. Interactive Exploration

As we announced at the beginning of this section the interactive exploration mode requires the involvement of the designer in the exploration of the design space. Figure 5.10 depicts the situation and highlights

the specific changes with respect to the execution cycle we described previously in this chapter.

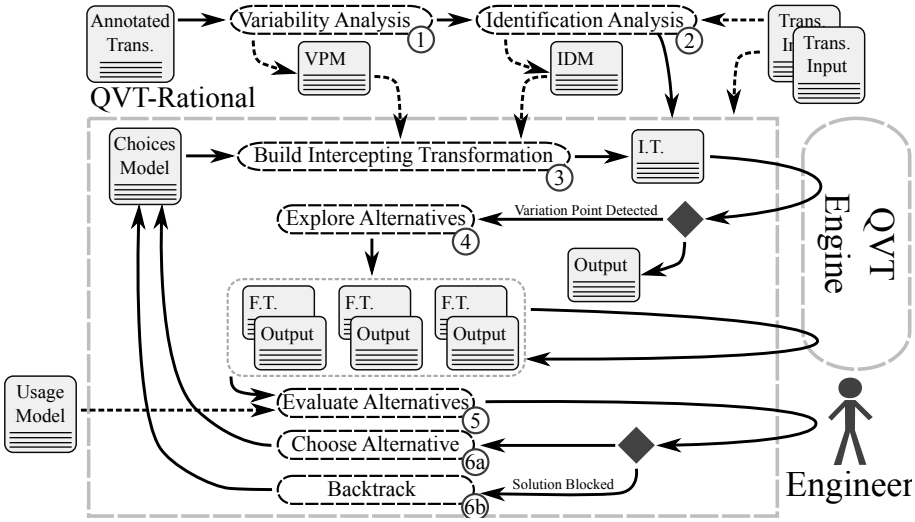

Figure 5.10.: The interactive execution cycle.

Two are the most important changes in the execution cycle. The first and most evident one is the involvement of the designer after the occurrence of a variation point. During interactive execution, the engineer is presented with the set of possible variants (accompanied with quality predictions and with the results of requirements evaluation) and is responsible for selecting which of the variants is the most promising (with respect to quality) and should be explored. As a consequence, in interactive exploration only one branch of the exploration tree is navigated at a time, no spawning of parallel and independent exploration cycles happens.

The other major difference in the process is the presence of what we call Fixing Transformations (FTs) which are responsible for generating complete system variants after the occurrence of a variation points. MDQP prediction techniques usually require complete system models in order to perform their computations, and the system designer needs access to that information in order to take a decision about which variant should be picked. For example, let us consider again the exploration tree of Figure 5.6. When the variation point corresponding to node 1 occurs, we need to evaluate the quality of candidates 2 and 3 in order to enable the designer in taking a conscious decision about which alternative to pick.

Fixing Transformations are responsible for generating complete sys-

tem variants after a variation point occurs. How fixing transformations are concretely performed depends on the underlying language (QVT-Operational or QVT-Relations) used to express the quality-driven model transformation, however, from an abstract point of view the mechanism is the same. Whenever a variation point is intercepted (i.e., a new variability for which no previous decision exists is identified) fixing transformations do not notify such event and do not transfer control to QVT-Rational, but instead they automatically select one of the viable alternatives. In practice, black-box interceptors are never invoked during a fixing transformations, but final system models are generated during each iteration over the exploration loop by picking random alternatives. Note however that alternatives are consistently selected across *FTs* for all those variation points which miss a decision of the engineer: for each variation point missing a decision the same alternative is picked up. This ensures that generated models differ only for the variation point being explored, that evaluation results are consistent, and corresponds to a local search strategy (which may not be practical in every case, but has the advantage of giving results fast).

For Operational-based transformations, concrete execution of *FTs* is rather simple and the *decideForVariationPoint* function that we weave in the *when* clause of the mappings implementing the variants takes care of this. During the execution of *FTs*, the *decideForVariationPoint* function works differently from its usual behavior: when a decision exists for a particular variability it acts as usual, but when a variability misses a decision it always responds true. In practice, this ensures that the fall-back mapping (the black-box interceptor mapping) is never reached/invoked, and the first available alternative is always picked when decisions are not available.

For what concerns Relational-based transformations, the concrete execution of *FTs* is slightly more complicated. First a different *FT* must be prepared for each possible variant, then it must be executed. The HOT responsible for generating *FTs* works similarly to how *ITs* are generated. The substitution predicate differs only for the relation called for the fall-back case. Instead of invoking the black-box interceptor when no decision has been already taken, the predicate invokes the relation implementing one of the possible alternatives, which we automatically and randomly select. This substitution is performed for every predicate defining a variation point, and the consequence is that the black-box interceptor relation is never invoked, but final system models are generated during each cycle over the exploration loop. Listing 5.9 shows the Fixing Transformation generated for the variant corresponding to node 7 in Figure 5.6. The predicate invokes the selected variants for already explored

variabilities, while it invokes the default *ClassToSingleTable* variant in all the other cases.

Listing 5.9: Building Fixing Transformations.

```
1  relation ClassToTable
2  {
3    domain uml c:Class {...}
4    domain rdbms t:Table {...}
5    where {
6      if (c.namespace.name = 'SimpleModel' and
7        c.name = 'Person')
8      then
9        ClassToSingleTable(c,t)
10     else
11       if (c.namespace.name = 'SimpleModel' and
12         c.name = 'Item')
13       then
14         ClassToTableInPartition(c,t)
15       else
16         ClassToSingleTable(c,t);
17       endif
18     endif
19   }
20 }
```

Backtracking

When a variation point is intercepted and alternatives must be explored, it is possible that some of them are not compatible with previous decisions due to some constraints. For example, in the ORM example we defined an exclusion constraint between the random generation strategy for identifiers and table partitioning. If an engineer has opted in a previous decision for random generation, when the *UsePartitioning* variation point occurs the only viable alternative is mapping onto a single table. However, partitioning may be necessary to meet performance requirements, but the generation of such a solution would thus be prevented by previous decisions.

It is also common that certain non-functional properties are affected by different variabilities; the decisions about them are interdependent and should be considered as a whole. A mechanism to decide at the same time for all the variabilities affecting the same property would suffice to cope with this issue, but our execution cycle permits the exploration of only one variation point at a time. To avoid this kind of problems we provide a *backtracking* functionality. If the applicability of some variants

has been prevented due to constraints during the exploration/evaluation stages, we show such situation to the engineer who may in turn decide to backtrack. The same happens if we recognize that a variation point affects a non-functional property already impacted by a previous decision, we are able to do this by using the information embedded into the *impact* annotation. When backtracking occurs, we identify the previous decisions involved and we clear them without changing the other decisions. During the next iteration over the execution cycle, the engineer will thus be able to to select another solution for the unfixed variability, enabling the selection of the blocked solutions or the exploration of different combinations of interdependent variabilities.

5.5. Summary

In this chapter we described how quality-driven model transformations are actually executed by QVT-Rational in order to generate feedback in the form of complete system variants (accompanied with quality predictions and checked for satisfaction with respect to the designer requirement). We showed how executing such a transformation is akin to exploring the design space, and we demonstrated how bleeding-edge MDSD techniques such as High-Order Transformations can be used for this purpose. We also showed how our execution model reuses existing QVT interpreters to concretely execute the transformations, and how we support both interactive and automatic execution modes. In the next chapter, we will provide some more details about the implementation of our framework and we demonstrate its effectiveness by running two case studies.

6. Implementation and Evaluation

In this chapter we provide some details about the implementation of QVT-Rational and we show how our framework concretely helps designers in getting feedback by running two case studies. This chapter is structured as follows.

First we give a brief overview of the implementation of QVT-Rational and we provide pointers to download its source code and install it. We discuss about the MDSD technologies we used to provide user-friendly editors and support tools to define the models involved in the QVT-Rational approach: from the specification of quality metrics, to feedback rules, and finally to requirements. Then we describe how we provide runtime support, that is, we talk about the implementation of our execution engine for QVT-Rational quality-driven model transformations and the QVT interpreters we support for this purpose.

The last part of this chapter outlines instead the outcome we obtained by using QVT-Rational in the context of two case studies. First we describe the QVT-Rational experience with respect to the ORM example we used throughout this thesis, by asking for feedback both with the interactive mode and with the automatic mode. Then we introduce another case study, i.e., the component allocation case study. We describe the meta-models involved, the quality metrics we consider and how we estimate them, and we define which variabilities incur in such example. We then show the results of asking feedback and we show how QVT-Rational, even in the case of non-trivial modeling scenarios, is able to provide feedback in a reasonable amount of time.

6.1. Implementation

QVT-Rational has been implemented as an Eclipse Platform [34] plug-in and leverages all the facilities provided by the Eclipse Modeling Framework (EMF) [33] to provide a comprehensive and integrated environment to define quality-driven model transformations and to execute them to obtain feedback. A primary concern in the development of QVT-Rational has been usability. We believe that the outcome of this thesis should

not be only an abstract approach, but also an usable development environment which we wish to be adopted as much as possible in the near future. An idea of what we mean with usability is shown in Figure 6.1, which depicts the user-friendly UI to define all the parameters to launch our quality-driven model transformations and obtain feedback. The complete source code, the documentation, as well as the update sites to easily install our framework are indeed available on the QVT-Rational web-site at `http://code.google.com/p/qvtr2/` .

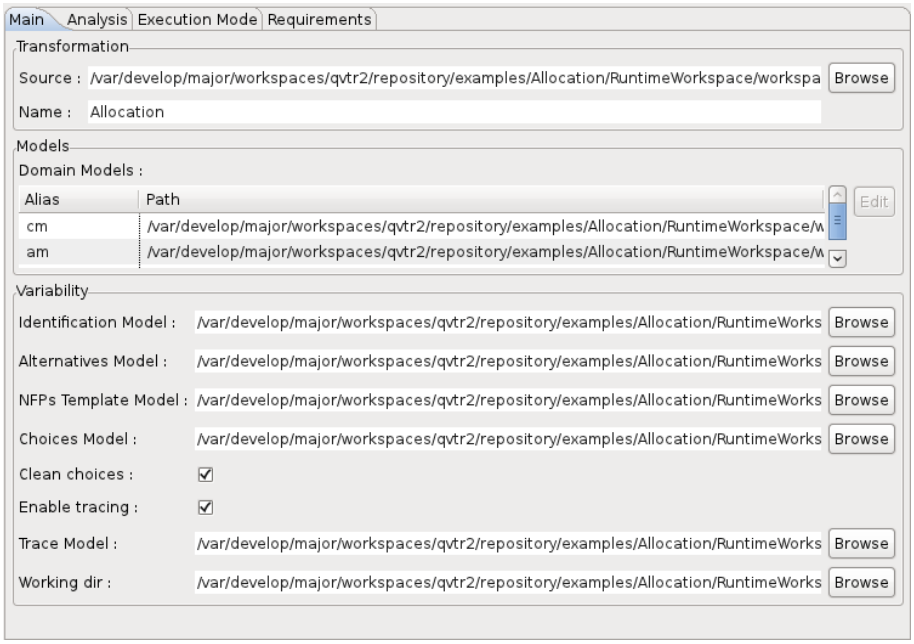

Figure 6.1.: An example of the user-friendly UI of QVT-Rational.

Eclipse provides its own modeling stack (i.e., EMF) to define meta-models and automatically generate boiler-plate code for editors and serialization. All the QVT-Rational meta-models we described in Chapters 4 and 5 have been developed with such de-facto standard in the MDSD community.

When talking about the definition of non-functional properties and about requirements, we used several times a user-friendly textual notation instead of graphical models. This textual notation has not been defined only for the sake of clarity in this thesis, but we concretely developed it and the associated tools to ease the interaction of QVT-Rational with designers and domain experts. Figure 6.2 depicts a screen-shot of the editors provided as part of the QVT-Rational framework. In detail, the textual notations and the associated editors have been defined

and automatically generated by using XText [36], a framework to easily create textual DSLs.

```
import nfps : "allocation.nfps";

requirements SimpleRequirements :
    req isScheduled {
        $Allocation.Schedulability(_) >= 1
    }

    req isGloballyScheduled {
        $Allocation.GlobalSchedulability >= 1
    }

    req isReliable {
        $Allocation.ComponentReliability(_) >= 0.85
    }

    req isGloballyReliable {
        $Allocation.GlobalReliability >= 0.85
    }

    req isClustered {
        $Allocation.Clustering(_) >= 1
    }

    soft req humanCost {
        $Allocation.Cost <= 30
    }
```

Figure 6.2.: Screen-shots of the textual DSL editors of QVT-Rational.

The model transformations and, in particular, the various transformations we outlined to analyze and to prepare quality-driven model transformations for execution, have been developed with the QVT-O language and our framework leverages the Operational interpreter for Eclipse provided as part of the M2M [108] initiative. We used the components defined in the same project also to parse both Operational and Relational transformations, and extract the EMF-based AST model which we modify with our preparation transformations.

Concerning the concrete execution of quality-driven model transformations, our implementation leverages the Eclipse interpreter we just mentioned to perform QVT-O transformations, while for Relational transformations we leverage the ModelMorf [102] engine provided by the Tata corporation. The Eclipse M2M initiative only partially supports the declarative part of the QVT specification; in detail, only parsers and editors are available for it while interpreters are still missing. Fortunately, two independent execution engines have been developed for QVT-Relations: the open-source Medini-QVT [101] interpreter provided by IKV++ Technologies and the already mentioned ModelMorf. We reviewed both and we decided to opt for ModelMorf [102] both for its reliability and completeness (Medini-QVT lacks in fact the implementation of several language features).

6.1.1. Integrating MDQP Tool-chains

An aspect that needs to be clarified here while discussing about the implementation is how the MDQP tool-chains, necessary for our approach to evaluate design alternatives, can be integrated in QVT-Rational. We recall from Section 3.2 that the ability to easily integrate existing analysis tools is an important and mandatory goal for QVT-Rational, and we will show this in the component allocation case study by using and integrating an existing MDQP tool-chain (i.e., the Klapersuite tool-chain [1]) to obtain predictions for reliability.

QVT-Rational is implemented with the Java programming language and, as a matter of fact, also the binding with MDQP tool-chains must be specified in the same language. Binding analyzers is however simple, it is in fact just a matter of creating the glue code to let QVT-Rational invoke the tool-chain and translate the results produced by the bound analyzer in a format suitable for consumption by our framework. In detail, the interface specifying the interaction protocol that must be implemented to bind MDQP analyzers is depicted in Listing 6.1. Two are the methods that must be supported: the *canBeDistributed* method which specifies if several instances of the same tool-chain can be run in parallel in order to speed-up the feedback provisioning process (if true, QVT-Rational will take care of all the extra burden of synchronizing with the various instances, nothing more is required on the analyzer side), and the *runAnalysis* method which QVT-Rational invokes to actually obtain predictions. The tasks performed by the *runAnalysis* method are usually simple, if the MDQP analyzer already follows the *augment with performance information and transform* approach. An hyphotetical implementation must in fact only translate the design models passed by QVT-Rational via method arguments to the specific formalism requested by the analysis technique, run the analysis, and translate the predictions in a format (i.e., a model compliant with the *NFPs* meta-model) compliant with QVT-Rational. Our experience in binding the KlaperSuite [1] MDQP tool-chain for the component allocation case study is an explanatory example of the effort required to write this glue code. The binding consists of approximately five hundred lines of code and required two days of work for a user with average experience about QVT-Rational and about KlaperSuite.

6.2. The ORM Case-study

In this section we demonstrate our approach for feedback provisioning by running the ORM case study both in manual and in automatic mode.

Listing 6.1: The interface to bind MDQP tool-chains.

```
public interface IAnalysisExecutor {
  /**
   * Checks whether this executor can be distributed
   * and executed remotely.
   * @return true if the executor can be distributed,
   *   false otherwise.
   */
   boolean canBeDistributed();

  /**
   * Concretely runs the analysis.
   * @param inputModels
   *        the input models for the analysis,
   *        cannot be null.
   * @param properties
   *        the analysis properties, cannot be null.
   * @param logger
   *        the logger, cannot be null.
   * @param token
   *        a cancellation token to signal analysis
   *        cancellation, cannot be null.
   * @return the results of the analysis.
   * @throws IllegalArgumentException
   *        if some of the non nullable arguments
   *        are null.
   * @throws AnalisysException
   *        if the analysis failed.
   */
  List<IPath> runAnalysis(List<IPath> inputModels,
    Map<String,String> properties, IStreamLogger logger,
    CancellationToken token)
    throws IllegalArgumentException, AnalisysException;
}
```

The results we show for the manual case refer to both the Operational-based transformation and to the Relational-based transformation. For what concerns instead automatic exploration, we used only the QVT-O-based transformation. As it will be clear next and, in particular, when discussing the results, the Modelmorf interpreter we use to execute QVT-R transformations is extremely slow, and this poses serious limitations when automatic exploration is used to get feedback with Relational-based transformations. This is not the case for the QVT-Operational counterpart, where execution is fast and represents only a small amount of the time required to get feedback.

As we outlined in Chapter 3, once the MDSD environment has been

prepared, four are the inputs required by QVT-Rational in order to be able to obtain feedback: a quality-driven model transformation extended with annotations to generate feedback (which we already described), the *Input Models* for the transformation (which in our case are the Class Models representing the domain entities of interest), a *Usage Model* describing the environmental conditions to perform quality analyses, and a *Requirements Model* specifying when found design solutions should be considered valid with respect to the designer's willings.

To show that QVT-Rational is able to handle also non-trivial MDSD scenarios, the input model we adopted has been taken from [109], where the author considers several open source and commercial e-commerce systems in order to extract a common architecture, to identify variabilities, and to model them as features.

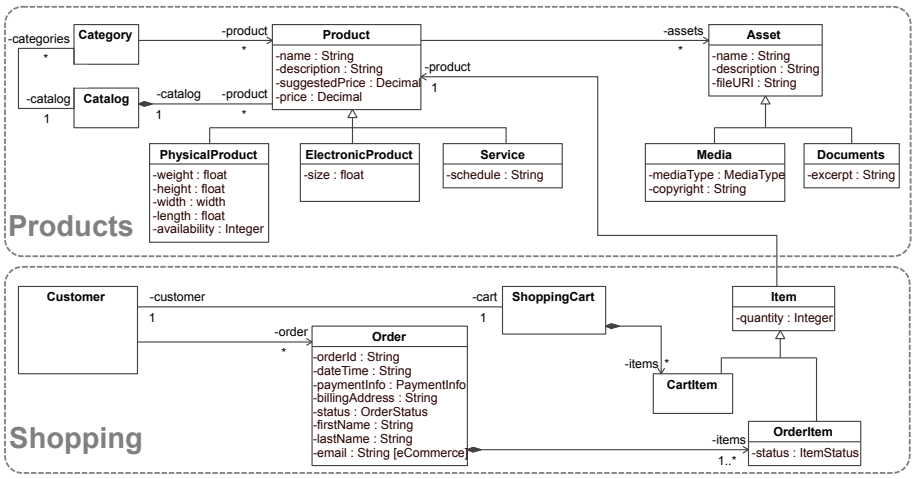

Figure 6.3.: The e-commerce domain model.

We ignore the part concerning feature parametrization and we concentrate on a subset of the domain and service models. Figure 6.3 shows the part of the application domain model we refer to in our example, which will be used as the input for our ORM transformation. Some details have been omitted for the sake of conciseness and clarity, the full specification of the example is however available in [109].

The input model we consider concerns the *products* and the *shopping* functionalities. The former provides the domain concepts to represent *product*s and to classify them, the latter defines the classes necessary to represent *shopping cart*s and *order*s. This portion of the e-commerce application is interesting from our perspective since it presents several points in which the variabilities defined for the ORM case study apply.

For example, the three inheritance hierarchies — for *product*s, for *asset*s, and for *item*s — will participate in the inheritance mapping variability. Moreover, as we will explain in the next section, the classes we consider in our example are of great importance for many functionalities of an e-commerce system. The way they are mapped onto the underlying database schema has a great impact on the non-functional properties of the system, and this motivates even more the need for guidance in applying techniques, such as partitioning, to meet quality-related requirements.

6.2.1. Requirements and Usage Model

The *Usage Model* and the non-functional requirements complete our evaluation scenario. The former is part of the QVT-Rational input, it is specified by the system designer, and it defines the conditions under which variants will be evaluated while performing the transformation; the latter are necessary to determine which variant should be considered valid during the feedback provisioning process and are specified by the end-user of QVT-Rational (i.e., the system designer).

To derive a reasonable *Usage Model* and requirements we concentrated on three important functionalities of an e-commerce system (i.e., *product finding*, *cart management*, and *placement of orders*) and we considered their top-most invoked queries (listed in the first column of Table 6.1). Such queries are business critical, they are subject to strict performance requirements and, finally, their performance is highly dependent on the structure of the database used to represent the domain entities they need to access. Detailed information about these two aspects is provided in Table 6.1 and in Table 6.2, where values have been selected from real e-commerce systems and modified to stress the functionalities of QVT-Rational.

Table 6.1 deals with queries and response time. Each row corresponds to one of the queries we consider for the evaluation and contains four values: the maximum response time allowed for the query, the workload we assume and its parameters, the domain entities accessed by the query, and the average number of memory pages accessed by the query for each entity. We recall from Section 3.3 that the QN-based technique we adopt computes service times from the number of memory pages accessed, for each table and by each query, and the time to access a memory page (which we assume 100 µs). For example, the response time of the *Find Product* (FP) operation should be at most 30 ms, it is supposed to be invoked with an arrival rate of 100 requests per second, and accesses all the classes belonging to the *Product*s inheritance hierarchy.

Table 6.1.: Usage profile and requirements for response time of queries.

Operation	Required Response Time [ms]	Wkld. Type	Arrival Rate [req/s]	Accessed Entities	# of Pages Accessed
Find Product (FP)	30	open	100	Product	200
				PhysicalProduct	160
				ElectronicProduct	20
				Service	20
Show Cart (SC)	30	open	40	ShoppingCart	1
				ShoppingCartItem	5
				Item	10
				Product	10
Add Item To Cart (AI)	30	open	10	ShoppingCart	1
				ShoppingCartItem	1
				Item	1
				Product	1
Place Order (PO)	30	open	2	ShoppingCart	1
				ShoppingCartItem	5
				Item	10
				OderItem	10
				Order	1
				Product	10
				PhysicalProduct	8

Table 6.2 deals instead with the frequency with which class instances appear with respect to the instances of the other classes participating in the type hierarchy. This information is required to predict the wasted space (for which we require 0.2 as the maximum acceptable value), and may be estimated by designers by analyzing application specific data such as product catalogs or customer profiles. In our example, we estimated them by considering a medium-sized e-commerce shop selling both electronic goods and digital contents. Some values have been omitted from the table for the sake of clarity. For example, the frequency of

classes not involved in hierarchies has been omitted, since its value is 1. A frequency of 0 has been assigned to super-types when generalizations are covering, i.e., "*every instance of a particular general Classifier is also an instance of at least one of its specific Classifiers*" [23]. *Product* and *Item* hierarchies are examples of this situation: the common super-type has frequency 0 and the frequencies of all sub-type instances sum up to 1.

Table 6.2.: Usage profile and requirements for wasted space.

Class	Frequency
Product	0
PhysicalProduct	0.8
ElectronicProduct	0.1
Service	0.1

Assets	0.3
Documents	0.35
Media	0.35

Item	0
OrderItem	0.8
ShoppingCartItem	0.2

Max Wasted Space : 0.2

6.2.2. Manual Exploration

The first experiment we have run consisted in asking to the QVT-Rational feedback in manual/interactive mode, i.e., we play the role of the designer during the process by deciding which variants should be picked whenever a new variation point occurs in the transformation. The results we show here refer to the execution of both the Operational-based transformation and the Relational transformation, which do not change in case of interactive exploration.

The aim of this experiment is to show that even a designer without strong experience about database optimization, performance, and Queuing Networks is able to identify a mapping solution which meets the specified non-functional requirements. If we shift ourselves in the opposite situation, that is, we act as experienced engineers, it is relatively easy to understand that the *Product* entity is the most-critical resource

from the perspective of the considered queries. By manually building a QN for the database schema corresponding to the input domain model, it is not difficult to identify that service centers for *Product*-related tables are the bottleneck, and that partitioning (which corresponds to incrementing the number of service centers [98]) is a valid solution to meet the required quality. The same applies to mapping of hierarchies and wasted space: the technique described in Section 3.3 can be manually applied.

Running the Transformation

Figure 6.4 summarizes the execution of the ORM case study, by representing with a tree the various decisions taken by the engineer during the entire process. The nodes of the tree represent either the occurrence of a variation point (shown as filled geometries annotated with information about the involved objects inside brackets) or models generated during the exploration of a variation point (shown as circles annotated with information about the corresponding alternative). Selected models/variants are connected with a solid edge to the next occurred variation point; leaf models are instead discarded during the process. For example, variation point *a* concerns the mapping of the *Asset* hierarchy; model 1 corresponds to the solution generated by using the *Joined Subclass* strategy, while model 2 corresponds to the solution adopting the *Upward Collapse* strategy. The selected solution is the latter: leaf model 1 is discarded while model 2 is connected to the next occurred variation point.

Table 6.3 (which is split on two pages and continued in Table 6.4 for space reasons) complements Figure 6.4 by providing detailed information about the feedback generated during the execution of the case study. Each row in the table is related to a specific variation point and to a specific model defined in the decision tree; it thus drives the decisions about which variant should be taken or discarded. Response time columns report the response time of each query as predicted by the QN-based analyzer invoked by QVT-Rational during the transformation; the last column shows instead the predicted wasted space. Table cells containing the not available (n.a.) symbol mean that the specific index is not affected by the considered variation point. For example, let us consider again variation point *a* and the rows corresponding to models 1 and 2. Estimates for the response time of all queries are not available, since *Asset*-related objects are not accessed by any considered query. The same applies for models 3-8 (which are all related to variation points involving *Item* classes) and *Find Product* (which does not access *Item* objects).

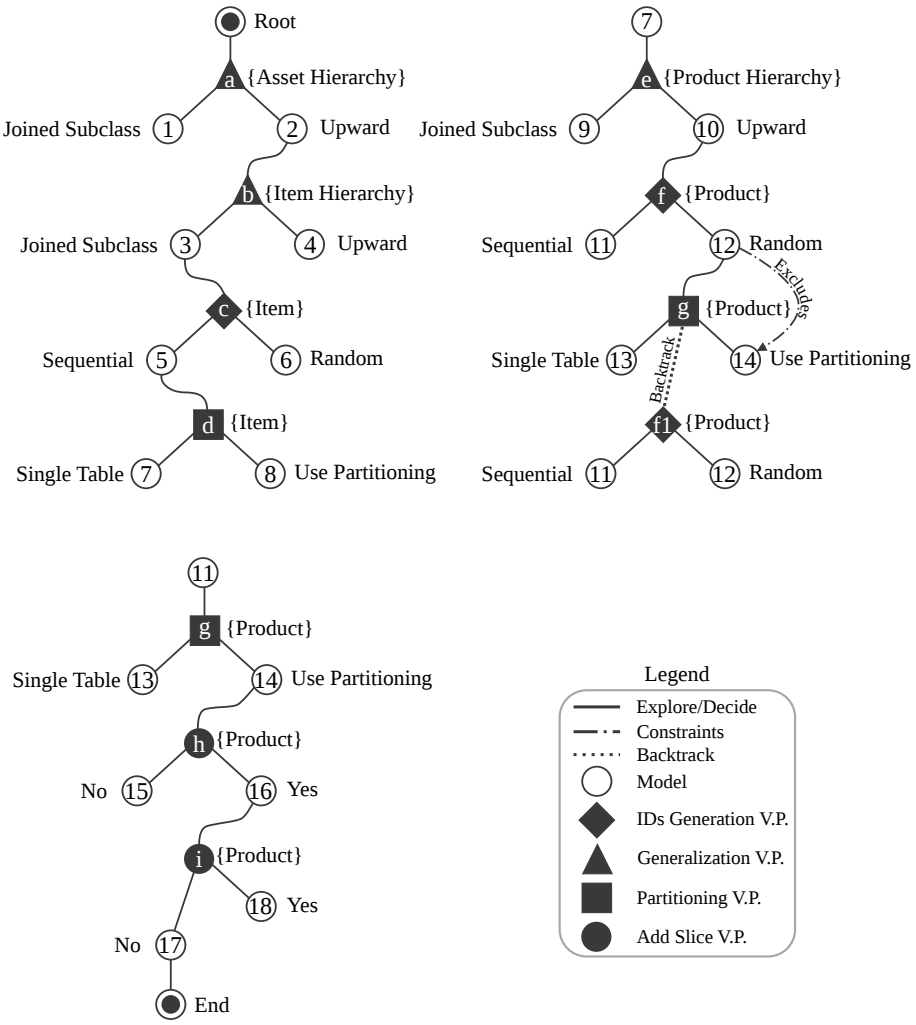

Figure 6.4.: The decision tree corresponding to the interactive exploration.

When values are available, table cells contain the predicted values for the corresponding index. For response times of queries, our system reports back both the global response time (*Sys*) and the single contributions of each entity involved in the variation point being explored. For example, let us consider variation points *b*-*d* (models 3-8). As shown by Figure 6.4, *Item*-related classes are involved by these variabilities. In Table 6.3 we report, for each query, both the global estimated response time and the contribution of the *Item* class (I), of the *CartItem* class (CI), and of the *OrderItem* class (OI). For space reasons, when both

Table 6.3.: Feedback information.

Var. Point	Model	Response Time [ms]				Wasted Space
		Find Product	Show Cart	Add Item	Place Order	
a	1	n.a.	n.a.	n.a.	n.a.	0
	2	n.a.	n.a.	n.a.	n.a.	0.1612
b	3	n.a.	I = 1.151 CI = 1.202 Sys = ∞	I = 0.429 CI = 0.430 Sys = ∞	I = 1.147 CI = 1.189 OI = 0.096 Sys = ∞	0
	4	n.a.	I = 1.152 CI = 1.197 Sys = ∞	I = 0.426 CI = 0.422 Sys = ∞	I = 1.161 CI = 1.192 OI = 0.094 Sys = ∞	0
b	5	n.a.	I = 1.155 CI = 1.201 Sys = ∞	I = 0.433 CI = 0.433 Sys = ∞	I = 1.150 CI = 1.195 OI = 0.098 Sys = ∞	0
	6	n.a.	I = 1.155 CI = 1.201 Sys = ∞	I = 0.431 CI = 0.427 Sys = ∞	I = 1.160 CI = 1.185 OI = 0.098 Sys = ∞	0

the global response time and the individual contributions are huge, the corresponding table cell contains the ∞ symbol. This is the case for variation points *e-h* (models 9-15).

Variation points *e-i* (models 9-18) refer to *Product*-related classes and need a special mention. *Product* classes (indicated with *P* in the table) are in fact the most critical. All queries read/write such entities and require access to a high number of records; impact on response time is thus high and the predicted value is infinite (rows corresponding to models 9-15). When this kind of situations happen, the feedback provided to the engineer may not convey enough information to take a correct

Table 6.4.: Feedback information continued.

Var. Point	Model	Response Time [ms]				Wasted Space
		Find Product	Show Cart	Add Item	Place Order	
d	7	n.a.	I = 1.154 CI = 1.199 Sys = ∞	I = 0.431 CI = 0.433 Sys = ∞	I = 1.151 CI = 1.194 OI = 0.099 Sys = ∞	0
	8	n.a.	I = 1.145 CI = 1.196 Sys = ∞	I = 0.423 CI = 0.419 Sys = ∞	I = 1.147 CI = 1.176 OI = 0.093 Sys = ∞	0
e	9	∞	∞	∞	∞	0
	10	∞	∞	∞	∞	0.1338
f - f1	11	∞	∞	∞	∞	n.a.
	12	∞	∞	∞	∞	
g	13	∞	∞	∞	∞	n.a.
	14	∞	∞	∞	∞	
h	15	∞	∞	∞	∞	n.a.
	16	P $\approx$ Sys = 29.4	P $\approx$ Sys = 12.8	P $\approx$ Sys = 10.3	P $\approx$ Sys = 13.0	
i	17	P $\approx$ Sys = 28.9	P $\approx$ Sys = 12.6	P $\approx$ Sys = 10.2	P $\approx$ Sys = 13	n.a.
	18	P $\approx$ Sys = 21.9	P $\approx$ Sys = 5.1	P $\approx$ Sys = 2.4	P $\approx$ Sys = 5.2	

decision. Let us consider variation point f: both the *sequential* (model 11) and the *random* strategy (model 12) exhibit the same infinite values for the response time prediction and may be considered as equivalent.

As a consequence, an engineer may be fooled and may select the wrong alternative (model 12). We recall from Section 3.3 that the *random* generation strategy conflicts with the usage of *partitioning*. The only viable option for variation point *g* is then using a *single table* (model 13) which, as shown by the corresponding row in Table 6.3, exhibits an infinite response time. Backtracking helps however recovering from these situations, and Figure 6.4 shows this. During exploration of variation point *g*, we signal that a previous decision blocks a viable alternative, we unlock the decision taken for variation point *f*, and we reconsider it during the evaluation of variation point *f1*. As shown in the decision tree, this time the *sequential* strategy (model 11) and *partitioning* (model 14) are selected. This in turn leads to the occurrence of variation points *h* and *i* that deal with deciding how many slices should be used in the partition. Table rows corresponding to models 16 and 17 show that three slices are sufficient to meet the 30 ms requirement, model 17 corresponds then to the final solution.

6.2.3. Automatic Exploration

We also asked to QVT-Rational to automatically explore the design space and generate feedback. In this case, we show only the results concerning the execution of the Operational-based ORM transformation defining the feedback rules. As we mentioned before, at the time this thesis has been written, the Modelmorf QVT-R interpreter is rather slow and getting feedback with automatic exploration is unfeasible. We have also run the same experiments we show here for the QVT-R transformation, but the results we obtained are useless and discouraging. To retrieve the same models identified by the QVT-O transformation in a reasonable amount of time, the execution of the quality-driven model transformation took nearly 27 hours, and the 98% of the time was spent generating system variants, not evaluating them as we should expect.

Getting back to our experiment with the QVT-O transformation, we asked QVT-Rational to generate the first 5 database schemas exhibiting a maximum 0.2 wasted space factor and a response time for each query not exceeding the 30ms threshold. The fact that we did not ask for every compliant model should not be a surprise. It would not be in fact useful from the designer perspective to obtain a huge list of possible system variants compliant with the non-functional requirements. Feedback provisioning is about giving useful suggestions, and providing a lot of possible solutions does not fit in this setting. It would in fact require a huge amount of work from the designer-side to evaluate them and select which one should be picked. If few alternatives are provided

as the outcome of the feedback process, these tasks become manageable and feedback provisioning useful.

In order to show the impact on the total execution time of quality prediction stages, we used two different implementations for the QN analyzers: one based on queuing network simulation and one based on Mean Value Analysis (MVA) [52]. The analyzer for wasted space is instead unique and has been ad-hoc developed.

Concerning the results, table 6.5 shows some statistics about the performance of QVT-Rational. All the experiments have been executed on a high-end workstation, equipped with a four core Intel i7 processor and 6 GB of memory. The T_i columns show the time required to produce the first i alternatives. Although the number of valid solutions is high, i.e., 1944 solutions out of 5836 viable alternatives, QVT-Rational is able to find variants and produce feedback in a reasonable amount of time. Results are even better when the MVA solver is used instead of the queuing network simulator. The time required to analyze the quality of found solutions drops to the 17% of the total execution time, and QVT-Rational produces feedback in 18 s.

Table 6.5.: ORM case study performance statistics.

Experiment	Solver	Solutions (Valid)	T_1 [s]	T_5 [s]	Analysis Time [s]
1	Simulator	5836(1944)	112.46	486.86	76%
2	MVA		7.21	18.53	17%

6.3. The Component Allocation Case-study

The other case study we adopt to show how QVT-Rational addresses the feedback provisioning problem is a well-known software engineering problem related to embedded systems: component allocation. Generally speaking, component allocation deals with the allocation of a set of software components over a set of limited hardware resources. This problem usually recurs while developing embedded systems — where it is exacerbated to its maximum given the very limited amount of hardware resources usually available — but it shows up also in other engineering fields, such as enterprise systems, where hardware limits are not a problem but where a certain QoS must be exhibited by the system and cost should be kept as small as possible.

The component allocation problem is rather interesting from the perspective of feedback provisioning. It is a case study adopted by other

works [14] in this research field, and in fact our case study is extracted from that work in order to compare our results with it and with respect to other approaches (Alloy [110] and SModels [111]) cited in it. Nonetheless, given a set of software components and of hardware resources, different possible allocations are usually viable but they exhibit different non-functional properties. For example, how components are allocated has an impact on the schedulability of operations performed by components, on the system reliability, on the performance and, finally, on the manufacturing cost. Hence, this example fits very well our variability and quality needs to show QVT-Rational in action.

We anticipate here that this case study aims at showing how QVT-Rational is able to automatically explore the design space in a reasonable amount of time also in non-trivial modeling situations. As a matter of fact, we specify the quality-driven model transformation responsible for generating feedback only with the Operational-based model transformation language. As we demonstrated while talking about the ORM case study, existing QVT-Relations interpreters are too slow to be of practical use for automatic exploration of the design space.

In the following we proceed as we did for the ORM case study in Section 3.3.1. First we describe the DSLs necessary to determine all the facets relevant for the component allocation problem, from the meta-models to specify software components and operations, to the meta-models to specify hardware resources, allocation, and information about the usage profile. Then we dig into the quality metrics we consider and the MDQP techniques to estimate them from design models. Finally, we outline the structure of the quality-driven model transformation responsible for generating viable allocations exhibiting a satisfactory QoS.

6.3.1. The Meta-models

The MDSD environment for the component allocation case study comprises four different meta-models. Two of them (the *Components* meta-model and the *Hardware* meta-model) are intended to be used by the system designer to specify the software components and the hardware resources, respectively. The third meta-model (the *Allocation* meta-model) binds the previous two abstractions by specifying on which hardware resource each software component should be deployed and, as we will see in a short, the quality-driven model transformation for this case study produces as output models conforming to it. The fourth meta-model (the *Usage Profile* meta-model) is again intented to be used by the system designer and complements hardware and software models with the information necessary to evaluate the quality of allocations.

Figure 6.5 outlines the structure of the *Components* meta-model. The provided concepts are quite self-explanatory, as well as their interconnections. The software concerns of the system being designed are described by a *ComponentModel* constituted by possibly several *Module*s. *Module*s capture cohesive pieces of functionality, and are further decomposed into *Component*s each one specifying certain *Operation*s. In real systems, the operations provided by software are never self-contained, but to achieve the functionality for which the system has been thought they collaborate by invoking each other. We allow for the modeling of this aspect with the self-loop association defined over the *Operation* meta-class, which allows for the declaration of invocation relationships between *Operation*s.

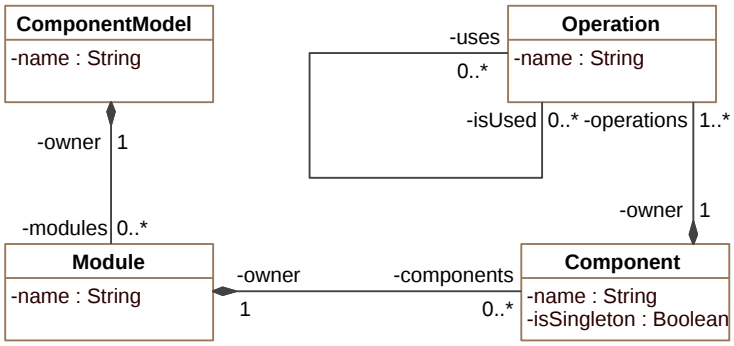

Figure 6.5.: The *Components* meta-model.

As it is outlined in Figure 6.6, the hardware resources available to deploy software components are specified inside an *HardwareModel*. We allow for the specification of both computation resources (i.e., the *Resource* and the *Cluster* meta-classes) and of communication links. Computation resources may be defined as self-contained entities or may be aggregated into *Cluster*s. Communication links may be instead of two types: *P2PLink*s represent point-to-point connections while *Bus*es represent multi-cast communication bridges over a set of computation resources, and are usually used to connect the *Resource*s defined inside *Cluster*s.

Figure 6.7 depicts the *Allocation* meta-model. This meta-model defines the kind of models which will be produced during the feedback provisioning process. An *AllocationModel* is a rather simple concept, it is a list of *Allocation*s linking software components to hardware resources, hence specifying how software and hardware can be composed in order to obtain a complete system design.

Figure 6.8 depicts instead the last meta-model involved in the component allocation case study. The *Usage Profile* meta-model allows for the

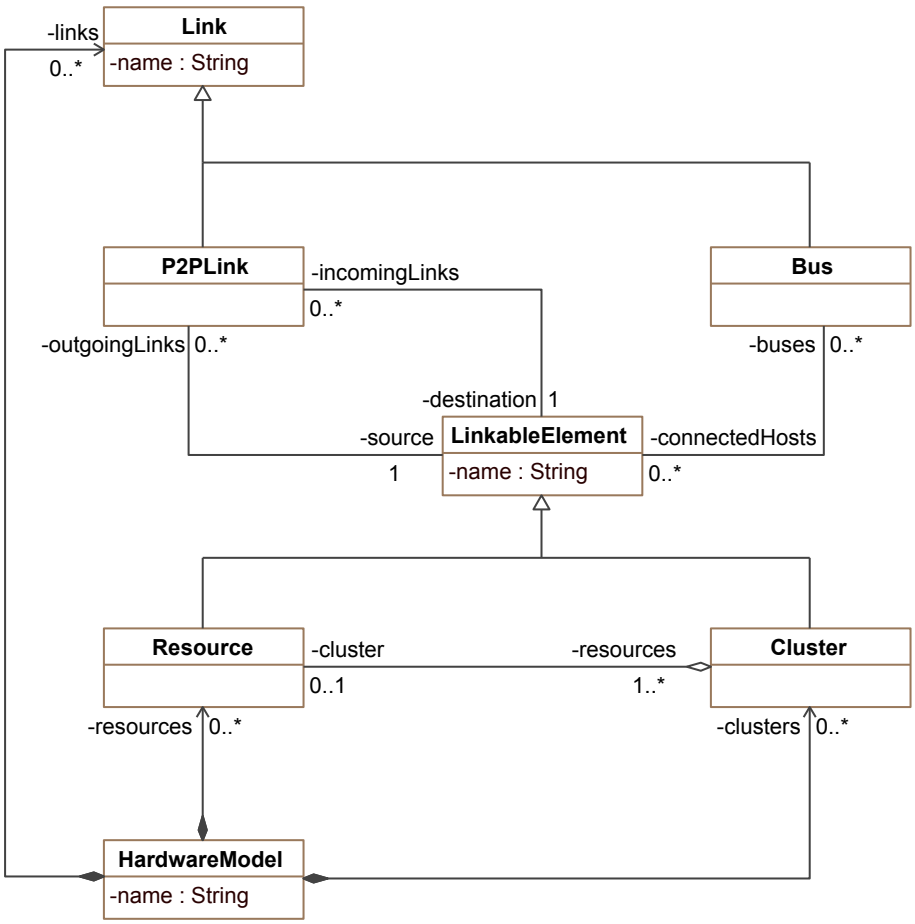

Figure 6.6.: The *Hardware* meta-model.

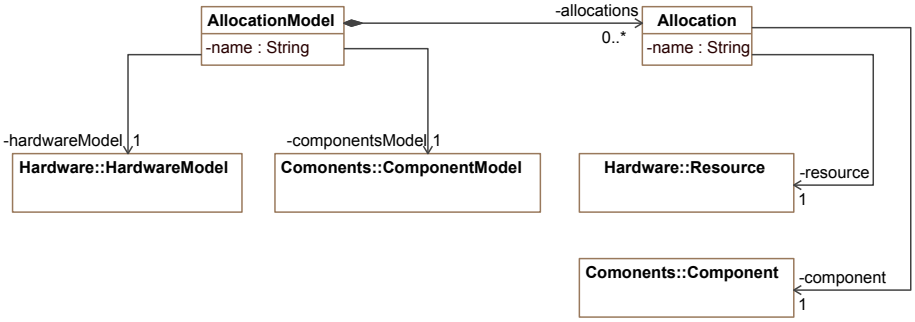

Figure 6.7.: The *Allocation* meta-model.

specification of the information (missing in the previous meta-models) necessary to perform the prediction of quality metrics. Instead of using a

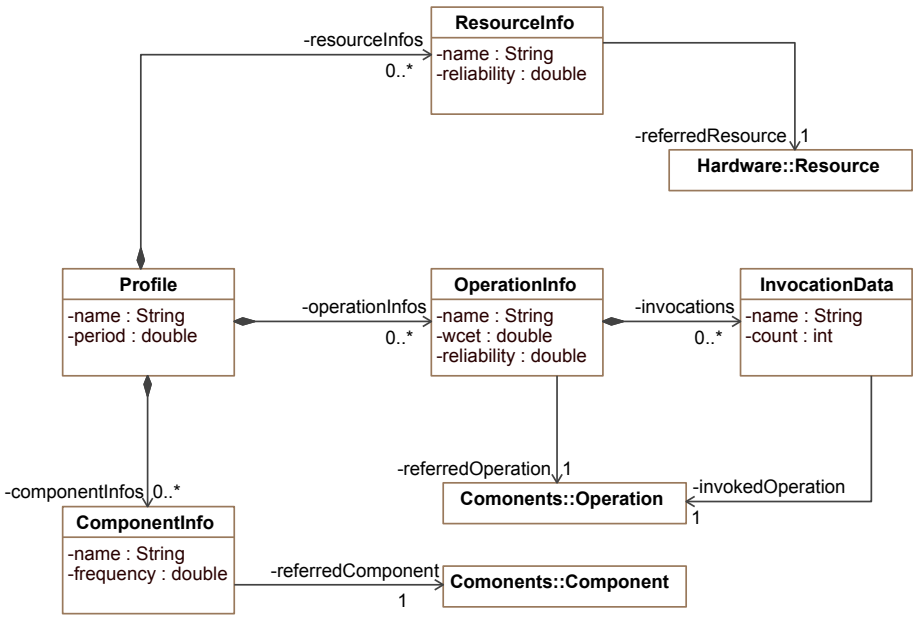

Figure 6.8.: The *Usage Profile* meta-model.

separate model to provide this kind of information, another option would have been to embed such information in the aforementioned abstractions. We indeed preferred to separate each individual concern in a dedicated model, a solution which also simplifies the process of examining what happens to a designed system when only quality concerns change. In detail, the *Profile* model allows for the specification of quality aspects concerning hardware resources, software components, and provided operations via the **Info* meta-classes. The *ResourceInfo* meta-class enables the specification of reliability information about hardware resources via the *reliability* attribute. The *OperationInfo* and the *ComponentInfo* meta-classes allow instead for the specification of information about reliability and performance for *Operation*s and *Component*s, respectively. Reliability of software can be specified with the same-named attribute for the *OperationInfo* meta-class. The specification of performance-related information is instead scattered across several concepts. In particular, whenever an *Operation* requires the invocation of another *Operation* to complete its computations, the number of times such other operation is invoked can be specified via the *InvocationData* meta-class and its *count* attribute. This information, as we will see in the next subsection, contributes to the estimation both of the system reliability and of the system schedulability property. In an embedded system there are usually some operations which are invoked periodically, for example sensing or

actuation operations. In order to correctly model also this aspect (which concurs to the prediction of the schedulability quality property) we allow for the specification of a *period* in the *Profile* meta-class, of the *frequency* with which each *Component* invokes its operations during each period, and of the *Worst Case Execution Time* (*wcet*) for each operation.

6.3.2. Quality Metrics and Prediction

Four are the quality metrics we consider in the allocation case study: reliability, schedulability, clustering, and cost. Listing 6.2 shows in detail the metrics we define with the textual notation for the *NFPs* meta-model provided by QVT-Rational. For each metric we define both a global property (computed globally with respect to the system design) and local properties (computed with respect to *Component*s and possibly *Operation*s). In the following we discuss them in detail.

Listing 6.2: Quality metrics for the allocation case study.

```
1  import componentsMM : "ComponentsMM.ecore";
2
3  PropertyModel Allocation {
4          template Schedulability {
5                  context component[Component];
6          }
7          template GlobalSchedulability { }
8
9          template Clustering {
10                 context module[Module];
11         }
12         template GlobalClustering { }
13
14         template GlobalReliability { }
15         template ComponentReliability {
16                 context component[Component];
17         }
18         template OperationReliability {
19                 context component[Component];
20                 context operation[Operation];
21         }
22
23         template Cost { }
24 }
```

Lines 4-7 define the schedulability property, which deals with the ability to execute the operations defined in the software components given the hardware on which they are deployed. In practice, schedulability is a

boolean property telling if the final system resulting in the composition of hardware and software is valid, that is, if all the provided functionalities can be computed with respect to the timing constraints. We define both local schedulability, whose value will be computed with respect to a single component, and global schedulability, whose value will depend on the schedulability of all components. Schedulability depends on three factors specified in the usage profile of the system — the worst case execution time (*wcet*) of each operation, the frequency, and the period with which components invoke operations — and is estimated by performing simple arithmetic. In detail, checking if a system is schedulable corresponds to computing the summation over the hardware resources of the times required by each operation to be executed and checking if the result is less or equal than the period. This computation is expressed by the following formula (which the analyzer declared in the quality-driven model transformation implements)

$$\forall r \in ress, \forall c \in compAllocation(ress) \Bigg(\sum_{op \in ops(c)} \Big(wcet(op) * freq(c) + \sum_{iop \in invoked(op)} (wcet(iop) * count(op, iop)) \Big) \leqslant period \Bigg) \tag{6.1}$$

where *ress* is the set of hardware resources, the *compAllocation* function returns the set of software components allocated on a specific hardware resource, the *ops* functions returns the set of operations defined by a component, the *invoked* function computes the set of operations invoked by another operation and the *count* function returns the number of times an operation is invoked by another operation.

Lines 9-12 of Listing 6.2 define instead the clustering property both in its global form and in its local form (referred to each software *Module* as declared by the property context). This boolean property deals with a well-formedness constraint about allocations: its value will be true if all the *Component*s declared in the same *Module* are allocated onto the same *Resource* or *Cluster*. Computing a value for this quality property is rather simple, it is just a matter of iterating over all the defined modules and checking that the allocation satisfies the constraint.

Lines 14-21 define instead reliability properties, both in their global and in their local form (computed with respect to *Component*s and *Operation*s). To compute reliability predictions we reuse the Markov Chain-

based analyzer developed for Klapersuite [1], an MDQP tool-chain for component-based software systems. Klapersuite (and the analyzer declared in the quality-driven model transformation as a consequence) uses a particular class of Markov Chains to perform the analysis. System models are translated in fact into Recursive Markov Chains (RMCs): a collection of finite-state Markov Chains with the ability to invoke each other, possibly in a recursive way. Recursive Markov Chains have been introduced in [112] and come with a strong mathematical support. RMCs can be analyzed (by means of equation systems) in a very efficient way in order to evaluate reachability properties. Reliability, intended as the probability of successfully accomplishing the assigned task, as well as the probability of failure given that the execution has reached a certain execution state, can be formalized as reachability properties, as well as a number of other interesting requirements.

Finally, line 23 defines the last quality metric, i.e., cost. This metric deals with manufacturing cost and is computed by performing simple arithmetic. We assume a unitary cost scheme (i.e., each hardware piece has the same cost equal to 1), hence computing it is just a matter of counting the number of resources (both computation resources and communication resources) used in the allocation.

6.3.3. Specifying Feedback

For the component allocation example, generating feedback consists in generating viable allocations (i.e., allocations which exhibit a certain quality level with respect to the metric we defined previously) given a component model and an hardware model. In practice, component allocation is a rather simple problem from a conceptual point of view: deployment is the only aspect that can vary. If we denote with C the set of components and with H the set of hardware resources, to generate all the possible deployments we proceed by iterating over the tuple set $T = C \times H$ and by finding all the possible subsets of T that map every component $c \in C$ on exactly one resource.

The quality-driven model transformation which generates viable allocation models works in this manner. Only one variability impacting all the defined quality metrics is defined in such a transformation, i.e., the variability which decides if given a specific component and a specific hardware resource allocation should take place or not. In this section we clarify this point and we concentrate on how we define such variability. For all the missing details, the interested author should refer to Appendix B where the complete transformation is listed and to the QVT-Rational website.

Listing 6.3 shows an important excerpt of the transformation responsible for generating feedback. The *allocate* contextual mapping declared on lines 1-5 accepts in input the list of available hardware resources (i.e., the *host*s) and, if the *Component* passed as the mapping context is not already allocated, iterates over the set of available hardware resources (the **foreach** on lines 6-21) and decides on which one it should be allocated (the **map** instruction on line 15).

Listing 6.3: The mapping invoking the variability.

```
1  mapping Component::allocate(
2    in availableHosts:OrderedSet(Resource))
3  when {
4    not self.isAlreadyAllocated();
5  } {
6    availableHosts->forEach(h) {
7      if (h = availableHosts->last())
8      then {
9        -- If this is the last available host
10       self.map defaultAllocate(h);
11       break;
12     } endif;
13
14     -- Otherwise decide to allocate or not
15     self.map allocateOn_VariationPoint(h);
16
17     -- Exit loop if component has been allocated
18     if (self.isAlreadyAllocated()) then {
19       break;
20     } endif;
21   };
22 }
```

The mapping invoked on line 15 which implements the *ComponentAllocation* variability is shown in Listing 6.4. The contexts of the variability are represented (as we outline in Chapter 4) by the arguments of the mapping (i.e., the contextual *Component* and the hardware resource), while the viable alternatives are two: either the transformation decides for the allocation or not. If the *Allocate* variant is selected, a new *Allocation* object representing this decision in the output model of the transformation is created, and the transformation proceeds by examining the next non-allocated component. As the *impact* annotation declares on lines 4-13, this variability has an impact on all the quality metrics we consider for this case study. This means that changing the allocation is the only possible way to improve or worsen the quality exhibited by the produced allocation model. Another interesting aspect of

the code shown in Listing 6.4 is the *requires* constraint declared on line 22. This constraint deals with the clustering property; it ensures that whenever a component (which is part of a *Module*) is allocated, the components defined in the same *Module* are allocated on the same hardware resource (which may be a simple resource or a cluster). This constraint is not necessary to obtain valid allocations, we may obtain the same effect by pruning non-clustered solutions by imposing a requirement over the schedulability property. It has however a nice side-effect on the performance of QVT-Rational while exploring the design space. The constraint in fact automatically excludes the generation of non-clustered solutions, i.e., they will not be generated, evaluated for quality, and time is saved during the feedback generation process.

6.3.4. The Experiments

The feedback we asked to QVT-Rational for this case study consisted in generating the first 5 acceptable deployments by running the system in automatic exploration mode. The QoS level requested by the allocation models produced by QVT-Rational is shown in Listing 6.5, which depicts the exact non-functional requirements we have specified for the case study. In detail, valid solutions should exhibit a limited cost (the requirement on lines 20-22, marked as soft in order to express a preference for the QoS level concerning other metrics and to consider cost only at the end), they should exhibit a reliability greater than 0.85 both globally and locally with respect to each component (the requirements on lines 8-14), and should be schedulable (the requirements on lines 4-10).

We recall that schedulability is a boolean property, either a system (or a component) is schedulable or not. However, our *NFPs* meta-model allows only the definition of numerical values for non-functional properties. In this case we adopted the usual convention when representing boolean values with integer numbers: 1 stands for true, while 0 stands for false. The requirements concerning schedulability follow this convention, i.e., we require only schedulable systems by requiring only system with a schedulability value grater of equal than 1.

We must indeed spend here some more words about the requirements we imposed, in order to explain why we impose a constraint both on the local and on the global version of the reliability metric and of the schedulability metric. Specifying only the global versions of these requirements would be sufficient to identify valid solutions, but it would have had bad consequences on the performance of QVT-Rational and, in particular, on the heuristic to guide the exploration of the design space. Local requirements have been specifically added to cope with this problem. In fact,

Listing 6.4: The Allocation Variability.

```
@varpoint {
 name := ComponentAllocation,
 analyzer := examples.allocation.Analyzer(),
 impact := {
  nfps::Schedulability($"self"),
  nfps::GlobalSchedulability(),
  nfps::Clustering($"self.owner"),
  nfps::GlobalClustering(),
  nfps::ComponentReliability($"self"),
  nfps::OperationReliability($"self", $"self.operations"),
  nfps::GlobalReliability(),
  nfps::Cost()
 }
}
mapping Component::allocateOn_VariationPoint(
  in host:Resource) disjuncts
  Component::dontAllocateOn_Variant,
  Component::allocateOn_Variant;

@variant {
  name := Allocate,
  requires := Allocate($"self.module.components", $"host")
}
mapping Component::allocateOn_Variant(in host:Resource) {
  getAllocationModel().allocations +=
    object allocation::Allocation {
      name := self.name + ' on ' + host.name;
      component := self;
      resource := host;
    };
}

@variant {
  name := DontAllocate
}
mapping Component::dontAllocateOn_Variant(
  in host:Resource) {
  -- Just do nothing
}
```

when the local versions of these requirements fail, they carry information about which component was involved in the failure: our system is thus able to identify the variabilities involving such component and schedule with higher priority the navigation of the exploration tree branches more likely to immediately solve the schedulability/reliability problem for that specific component. This kind of information is not carried by the global

Listing 6.5: The requirements for the component allocation case study.

```
1  import nfps : "allocation.nfps";
2
3  requirements SimpleRequirements :
4  req isScheduled {
5    $Allocation.Schedulability(_) >= 1
6  }
7
8  req isGloballyScheduled {
9    $Allocation.GlobalSchedulability >= 1
10 }
11
12 req isReliable {
13   $Allocation.ComponentReliability(_) >= 0.85
14 }
15
16 req isGloballyReliable {
17   $Allocation.GlobalReliability >= 0.85
18 }
19
20 soft req acceptableCost {
21   $Allocation.Cost <= 30
22 }
```

version of these requirements, and our experiments show that without local constraints the feedback process takes about 5 to 8 times the time required to generate feedback (with respect to the values shown in Table 6.6).

Getting back to the case study results, we have run 4 experiments under different conditions, that is, we have asked for feedback with four different input models. The input model for the first experiment has been manually developed, and resembles the architecture of a small sensing-reacting embedded device composed by three sensing modules, one computation module, and one reaction module which invokes the operations provided by the other components several times in a period. The remaining experiments have been instead generated randomly by specifying number of components, modules, and resources. This is also the practice adopted in [14] to show Formula in action. The interested reader can find the complete input models for the case study on the QVT-Rational web-site[1].

Table 6.6 shows some statistics about the measured performance of QVT-Rational. The T_i columns show the time required to produce the

[1] `http://qvtr2.googlecode.com`

first i alternatives, while the last column shows the amount of time taken by the MDQP tool-chain to predict quality metrics. All the experiments have been executed on a high-end workstation, equipped with a four core Intel i7 processor and 6 GB of memory. QVT-Rational is rather fast

Table 6.6.: Allocation case study performance statistics.

Exp.	Modules	Compo-nents	Resources	T_1 [s]	T_5 [s]	Analysis Time
1	2	5	5	8.17	34.79	6%
2	2	5	5	0.75	5.68	4.25%
3	3	10	10	2.36	40.29	4.39%
4	5	15	5	159.44	251.55	9.87%

for small and medium-sized case studies (i.e., experiments 1-3), while for the largest case study (experiment 4) the time to generate feedback increases but still remains reasonable from a designer perspective. This is even more encouraging if we consider that component allocation is an NP-Hard problem and if we compare with the results described in [14]. Formula is much faster in solving the same problem (given the logic program encoding) and outperforms Alloy [110] and SModels [111] in performing the same computation. However, similarly to our results, it shows exponential growth for little larger models. Indeed, it must be noted that the example has been extended with reliability (which was not covered in [14]), which is not possible in logic-based frameworks without the extra-effort to encode analysis tools as logic programs.

6.4. Discussion

In the previous sections we described the two case studies we have run to show the QVT-Rational approach for feedback provisioning about quality concerns. We showed how non-injective model transformations augmented with information about quality can be used to specify the feedback rules also in non-trivial MDSD scenarios, and we demonstrated how our execution engine is able to produce system variants compliant with the designers' requirements in a reasonable amount of time. What we have highlighted till now are however only the strength points of the QVT-Rational approach. For the sake of precision, we must discuss here also some of the drawbacks we recognize for our approach. In particular, four are the major drawbacks concerning our approach:

- **Non-Guaranteed Optimality.** In the previous chapters we showed

that quality-driven model transformations can be run both in automatic and in interactive mode. In interactive mode only some branches of the design space tree are explored and which ones are actually explored depends on the decisions of the designer involved in the feedback process. In automatic mode the design space is automatically explored and, although complete exploration is possible, navigating the whole space of solutions may not be feasible when models are huge. When either interactive or automatic (and limited) exploration modes are used, there is no guarantee that the found solutions are the best in terms of exhibited quality. For example, in our case studies, we asked for the first i solutions compliant with the requirements, but nothing guarantees that better solutions exist. In practice, it is very likely that the feedback produced by QVT-Rational consists in a local optimum, which may not be the best thing in every situation and which represents one of the most important drawbacks of QVT-Rational. We must indeed point out here that this problem is however not specific to our approach, but also other methodologies (especially the ones based on meta-heuristic techniques) suffer from it.

- **Efficiency.** When analysis tools require a lot of time to be executed, asking for feedback can be a time-consuming operation. This is also true when the input system models are large, several valid alternatives are available, and the complete exploration of the design space is requested. When such situations happen, an online usage of our system — i.e., while the designer is developing or modeling — is not practical. Blocking the designer for a long time to obtain feedback is not feasible. However, we must again point out here that this problem is not specific only to QVT-Rational, also other approaches exhibit performance problems. Nonetheless, QVT-Rational may still be used by running it in parallel with the designer or during moon-light hours.

- **Impact of Requirements on Guidance.** To improve the efficiency of automatic exploration, QVT-Rational relies on the information conveyed by requirements and by their evaluation to identify the most promising areas of the design space. As we showed while discussing the guidance heuristic, different requirements convey different kinds of information. The more precise (i.e., contextual) a requirement is, the more likely it is that the important decisions which should be changed will be identified. As a matter of fact, although this connection may not appear clear to non-experienced QVT-Rational users, there is a link between

how requirements are specified and the time required to generate feedback.

- **Memory and Storage Footprint.** Time is not the only resource important from the efficiency point of view. When describing the QVT-Rational execution model in the previous chapter, we highlighted that for each new branch discovered in the exploration tree a new independent exploration process is spawned. As a consequence of this, all the artifacts concerning the execution (e.g., the input design models or the QVT-Rational execution models) are copied for the new processes and this negatively impacts the used storage and memory resources. Although this may seem a non-negligible problem, we accounted for it in our implementation and the resources (artifacts and memory) for the already explored nodes in the design tree are freed as soon as possible. This positively decreases the amount of storage and memory required by our system which, in our experiments, amounted to a maximum of 67% of the resources available on the workstation we used to run the case studies.

7. Conclusions

In this thesis we described QVT-Rational, our multi-model, multi-target, programmable solution to provide design feedback about non-functional properties of software systems.

QVT-Rational is an holistic solution encompassing all the characteristic aspects of feedback provisioning. The idea behind our framework is to formalize and promote reuse of the deep knowledge possessed by domain experts about non-functional concerns, quality-related issues and solutions by means of quality-driven model transformations. QVT-Rational leverages then such information to generate suggestions to system designers (in the form of complete system variants satisfying certain requirements) in order to identify viable solutions when quality-related issues are identified in design models.

From the domain expert perspective, our framework provides all the instruments (e.g., tools, meta-models, languages) to define how a system can be modeled, what are the non-functional properties of interests, and how they can be evaluated by easily integrating existing MDQP tool-chains in the process. QVT-Rational provides its own language to define quality-driven model transformations, by extending with appropriate constructs and annotations the two transformation languages defined in the QVT specification.

QVT-Rational supports the system designer by providing a convenient language to formalize requirements about non-functional properties, and by providing an efficient engine to execute quality-driven model transformations in order to identify viable system variants exhibiting the required level of quality. We showed how the execution of quality-driven model transformations corresponds to exploring the space of the possible design solutions, and we described how QVT-Rational efficiently explores such space by using heuristics and information about non-functional requirements.

7.1. Obtained Results

To demonstrate the validity of our approach, we showed in the previous chapter how QVT-Rational is able to generate suggestions to system designers in a reasonable amount of time even in non-trivial modeling

situations. In particular, we showed how it is simple to define the rules to generate system variants, to bind them to quality aspects, and to use them to concretely generate alternative designs in the context of two well-known software engineering problems: Object-Relational Mapping (ORM) and component allocation. We demonstrated that, even when several variabilities exist and contribute to the quality of system designs, and even when big models are involved in the process, QVT-Rational is able to identify alternative solutions in a time compatible with the needs of software designers. This enables QVT-Rational to compete also with other existing frameworks to generate feedback.

Nonetheless, we should point out here that QVT-Rational also presents other advantages: first of all language uniformity (no ad-hoc languages must be mastered to define how feedback can be generated but we leverage existing formalisms well-known in the MDE community), second we are not limited to specific exploration directions or quality metrics but QVT-Rational is open in this sense, third we support in a seamless way the integration with existing MDQP tool-chains to predict the quality of system designs.

7.2. Future Work

Concerning future work, at the time this thesis has been written, we are moving in two opposite directions: shifting our approach also at run time and improving support in the specification of the feedback rules from the domain expert perspective.

Concerning the first direction, our intuition is that the very same techniques to provide feedback at design time could also be applied at run time to provide self-adaptation capabilities to software systems. In particular, we are exploring the possibility of navigating the design space under different environmental conditions and pre-compute the different system variants exhibiting the required QoS for every specific situation. At run time, when the environment changes, these pre-computed alternative may be then applied to the system (for example by reconfiguring it) in order to self-adapt.

Concerning the second direction, we highlighted in the previous chapters that specifying the feedback rules and how they are bound to quality concerns is anything but easy. In particular, identifying how each variability impacts quality (which metrics and how much) is difficult even for very experienced domain experts. In this sense, we are exploring the possibility to provide suggestions to domain experts in order to identify which quality metrics are impacted by each variability. Our idea

is to use techniques similar to the one described in [106], that is, build several variants of a complete system, monitor its real quality, and use reasoning and inference techniques to identify (which a certain degree of confidence) what is the binding between variability and quality.

7.3. Acknowledgements

The work described in this thesis has been mainly founded by the European Commission IDEAS-ERC Project 227977-SMScom.

A. The ORM Transformations

This appendix contains the complete listings of the transformations (both with QVT-Operational and with QVT-Relations) for the ORM case study.

A.1. The QVT-Operational Transformation

Listing A.1: The complete QVT-O transformation for the ORM case study.

```
modeltype simpleUml "strict" uses umlMM('umlMM');
modeltype simpleDb "strict" uses rdbmsMM('rdbmsMM');
modeltype ecore "strict" uses ecore(
 'http://www.eclipse.org/emf/2002/Ecore");
modeltype qvt "strict" uses qvtoperational(
 'http://www.eclipse.org/QVT/1.0.0/Operational");

@nfpmodel[nfps => "Uml2Rdbms.nfps"]

@keydefs{
 simpleUml::Package(#name),
 simpleUml::Classifier(#name, #namespace),
 simpleUml::Class(#name, #namespace),
 simpleUml::Attribute(#name, #owner),
 simpleUml::Association(#name, #owner),
 simpleUml::Generalization(#name, #owner),

 simpleDb::Schema(#name),
 simpleDb::Partition(#name, #schema),
 simpleDb::Table(#name, #schema),
 simpleDb::Column(#name, #owner),
 simpleDb::Key(#name, #owner),
 simpleDb::ForeignKey(#name, #owner)
}

transformation Uml2Rdbms_Replication(
 in classModel:simpleUml, out dbModel:simpleDb);

main() {
 var packages := classModel.rootObjects()[Package];
 assert fatal (packages->size() = 1) with log (
  "Input model does not contain exactly one package.");
 packages->map Package2Schema();
}

mapping Package::Package2Schema() : Schema {
 log("Mapping package " + self.name + " to schema");
 result.name := self.name;

 -- Map top classes
 result.table += self.classifier[Class]->map TopClassToTable();

 -- Map sub classes
 self.classifier[Class]->map SubclassToTable(result);

 -- Map associations
 self.association->map AssociationToRelationship(result);

 -- Decide for Paritioning
 self.classifier[Class]->select(c | not self.generalization->exists(g |
```

```
   g.subType = c))->map UsePartitioning_VariationPoint();
}

-- Marker mapping
mapping Class::ClassToTableMarker(inout table:Table) : Table {
 init { result := table; }
 log("Established marker mapping ClassToTable " + self.name +
   " ---> " + result.name);
}

mapping Class::ClassToPartitionMarker(inout partition:Partition) : Partition {
 init { result := partition; }
 log("Established marker mapping ClassToPartition " + self.name +
   " ---> " + result.name);
}

-- Base mapping for all ClassToTable mappings
mapping Class::ClassToTableActual(inout table:Table) : Table
{
 init { result := table; }
 result.name := self.name;

 -- Map all attributes
 result.column += self.attribute->map Attribute2Column();

 -- Map id attributes on autogen cols and primary key
 table.hasKey :=  object Key {
  name := self.name + "_pkey";
  self.attribute->select(a | a.isId = true)->forEach(attr) {
   var mappingCol := attr.resolveoneIn(Attribute::Attribute2Column)
     .oclAsType(Column);
   column += mappingCol;
  };
 };
}

mapping Class::TopClassToTable() : Table
when {
 not self.namespace.generalization->exists(g | g.subType = self);
} {
 log("Mapping top class " + self.name + " to table");

 -- Some basic checks
 assert fatal (self.attribute->exists(a | a.isId)) with log("Class "
   + self.name + " has no id attributes");

 self.map ClassToTableActual(result);
 self.map ClassToTableMarker(result);
}

mapping Class::SubclassToTable(inout schema:Schema)
when {
 self.namespace.generalization->exists(g |
   g.subType = self);
} {
 log("Mapping subclass " + self.name + " to table");

 -- Check that class participates to only one generalization as subtype
 var generalizations := self.namespace.generalization
   ->select(g | g.subType = self);
 assert fatal (generalizations->size() = 1) with log
  ("Class " + self.name + " has more than one super class");

 -- Get the table on which the superclass has been mapped
 var superClass := generalizations->any(true).superType;
 var superTable := superClass.resolveoneIn(Class::ClassToTableMarker);
 if (superTable = null) then {
  -- Super class has not been mapped, map it before
  superTable := superClass.map SubclassToTable(schema);
 } endif;
 schema.table += self.map InheritanceMapping_VariationPoint();
}

@varpoint {
 name := InheritanceStrategy,
 description := "Selects the strategy to map a generalization.",
 analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
 impact := {
  nfps::GlobalResponseTime(_),
  nfps::ResponseTime(_, $"self")
```

```
 }                                                                                  131
}                                                                                   132
mapping Class::InheritanceMapping_VariationPoint() : Table                          133
  disjuncts Class::AssociationCollapse_Variant,                                     134
            Class::UpwardCollapse_Variant;                                          135
                                                                                    136
@variant {                                                                          137
 name := UpwardCollapse,                                                            138
 description := "Maps a subclass on the parent table.",                             139
 excludes := AssociationCollapse(                                                   140
  $"self.namespace.generalization->select(g | g.superType =                         141
     self.namespace.generalization->select(g1 | g1.subType = self)                  142
       ->any(true).superType).subType")                                             143
}                                                                                   144
mapping Class::UpwardCollapse_Variant() : Table {                                   145
 init {                                                                             146
  var generalizations := self.namespace.generalization                              147
    ->select(g | g.subType = self);                                                 148
  assert fatal (generalizations->size() = 1) with log ("Class "                     149
    + self.name + " has more than one super class");                                150
                                                                                    151
  -- Get the table on which the superclass has been mapped                          152
  var superClass := generalizations->any(true).superType;                           153
  var superTable := superClass.resolveoneIn(                                        154
    Class::ClassToTableMarker);                                                     155
  result := superTable.oclAsType(Table);                                            156
 }                                                                                  157
                                                                                    158
 log("Mapping (UpwardCollapse) class " + self.name + " to table");                  159
                                                                                    160
 -- Invoke the marker mapping                                                       161
 self.map ClassToTableMarker(result);                                               162
                                                                                    163
 -- Map attributes                                                                  164
 result.column += self.attribute->map Attribute2Column();                           165
                                                                                    166
 -- Create the discriminator column, if not yet created                             167
 if (not result.column->exists(c | c.name = "discriminator")) then                  168
 {                                                                                  169
  result.column += object Column {                                                  170
   name := "discriminator";                                                         171
   type := "CHAR";                                                                  172
  };                                                                                173
 } endif;                                                                           174
}                                                                                   175
                                                                                    176
@variant {                                                                          177
 name := AssociationCollapse,                                                       178
 description := "Maps a subclass on its own table and                               179
   on an association with the table onto which the parent was mapped.",             180
 excludes := UpwardCollapse(                                                        181
   $"self.namespace.generalization->select(g | g.superType =                        182
      self.namespace.generalization->select(g1 |                                    183
     g1.subType = self)->any(true).superType).subType")                             184
}                                                                                   185
mapping Class::AssociationCollapse_Variant() : Table                                186
{                                                                                   187
 log("Mapping (AssociationCollapse) class " +  self.name + " to table");            188
                                                                                    189
 var generalizations := self.namespace.generalization->select(g |                   190
   g.subType = self);                                                               191
 assert fatal (generalizations->size() = 1)                                         192
   with log ("Class " + self.name + " has more than one super class");              193
                                                                                    194
 -- Get the table on which the superclass has been mapped                           195
 var superClass := generalizations->any(true).superType;                            196
 var superTable := superClass.resolveoneIn(                                         197
   Class::ClassToTableMarker).oclAsType(Table);                                     198
                                                                                    199
 result.name := self.name;                                                          200
                                                                                    201
 -- Invoke the marker mapping                                                       202
 self.map ClassToTableMarker(result);                                               203
                                                                                    204
 -- Create a primary key                                                            205
 result.hasKey := object Key {                                                      206
  name := self.name + "_pk";                                                        207
 };                                                                                 208
                                                                                    209
 -- Clone SuperClass ids for this class                                             210
```

```
 superTable.hasKey.column->forEach(cl) {
  result.hasKey.column += object Column {
   name := cl.name;
   type := cl.type;
   owner := result;
  };
 };

 -- Create a foreign key referring to the
 -- parent class pkey on this table pkey cols
 result.hasFKey += object ForeignKey {
  name := self.name + "_" + superClass.name + "_Gen";
  column := result.hasKey.column;
  refersTo := superTable.hasKey;
 }
}

mapping Attribute::Attribute2Column() : Column
{
 result.name := self.name;
 result.type := UmlTypeToDbtype(self.type.name);

 if (self.isId = true) then {
  self.map IdAttribute2AutogenColumn_VariationPoint();
 } endif;
}

@varpoint {
 name := AutogenerationStrategy,
 description := "Selects the strategy to auto
   generate values for id attributes",
 analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
 impact := {
  nfps::GlobalResponseTime(_),
  nfps::ResponseTime(_, $"self.owner")
 }
}
mapping
  Attribute::IdAttribute2AutogenColumn_VariationPoint() : Column
  disjuncts
    Attribute::IdAttribute2AutogenSequenceColumn_Variant,
    Attribute::IdAttribute2AutogenRandomColumn_Variant;

@variant {
 name := SequentialGeneration,
 description := "A sequential generation strategy."
}
mapping
  Attribute::IdAttribute2AutogenSequenceColumn_Variant() : Column
{
 init {
  result := self.resolveoneIn(Attribute::Attribute2Column)
    .oclAsType(Column);
 }
 assert fatal (self.isId = true) with log("Trying to map non id attribute "
   + self.name + " to an autogen column");
 result.autoGenerate := "SEQUENTIAL";
}

@variant {
 name := RandomizedGeneration,
 description := "A randomized generation strategy.",
 excludes := ClassToPartition($"self.owner")
}
mapping Attribute::IdAttribute2AutogenRandomColumn_Variant() : Column
{
 init {
  result := self.resolveoneIn(Attribute::Attribute2Column).oclAsType(Column);
 }
 assert fatal (self.isId = true) with log("Trying to map non id attribute "
   + self.name + " to an autogen column");
 result.autoGenerate := "RANDOM";
}

mapping Association::AssociationToRelationship(inout schema:Schema)
  disjuncts
    Association::OneToNAssociation,
    Association::NToOneAssociation,
    Association::NToNAssociation;
```

```
mapping Association::OneToNAssociation(inout schema:Schema)
when { self.sourceCardinality = 1 and self.destCardinality = -1; }
{
 log("Mapping association " + self.name);

 var sourceTable := self.source.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);
 var destTable := self.destination.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);

 assert fatal (sourceTable <> null) with log("Trying to map association "
   + self.name + " but source class has not been mapped");
 assert fatal (destTable <> null) with log("Trying to map association "
   + self.name + " but dest class has not been mapped");

 destTable.hasFKey += object ForeignKey {
  name := self.name + "_fk";
  refersTo := sourceTable.hasKey;
  sourceTable.hasKey.column->forEach(cl) {
   column += object Column {
    name := cl.name + "_" + self.name + "_fk";
    type := cl.type;
    owner := destTable;
   }
  };
 };
}

mapping Association::NToOneAssociation(inout schema:Schema)
when { self.sourceCardinality = -1 and self.destCardinality = 1; }
{
 log("Mapping association " + self.name);

 var sourceTable := self.source.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);
 var destTable := self.destination.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);

 assert fatal (sourceTable <> null) with log("Trying to map association "
   + self.name + " but source class has not been mapped");
 assert fatal (destTable <> null) with log("Trying to map association "
   + self.name + " but dest class has not been mapped");

 sourceTable.hasFKey += object ForeignKey {
  name := self.name + "_fk";
  refersTo := destTable.hasKey;
  destTable.hasKey.column->forEach(cl) {
   column += object Column {
    name := cl.name + "_" + self.name + "_fk";
    type := cl.type;
    owner := sourceTable;
   }
  };
 };
}

mapping Association::NToNAssociation(inout schema:Schema)
when { self.sourceCardinality = -11 and self.destCardinality = -1; }
{
 log("Mapping association " + self.name);

 var sourceTable := self.source.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);
 var destTable := self.destination.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);

 assert fatal (sourceTable <> null) with log("Trying to map association "
   + self.name + " but source class has not been mapped");
 assert fatal (destTable <> null) with log("Trying to map association "
   + self.name + " but dest class has not been mapped");

 var at:Table;
 schema.table += object at:Table {
  name := self.name;
  hasKey := object Key {
   name := self.name + "_pk";
  };

  -- Create a foreign key in the assoc table for the source table
  var sourceTableFKey := object ForeignKey {
```

```
    name := sourceTable.name + "_fk";
    refersTo := sourceTable.hasKey;
   };
   at.hasFKey += sourceTableFKey;

   -- Create a column in the assoc table for each source table id col
   sourceTable.hasKey.column->forEach(cl) {
    var newCol := object Column {
     name := sourceTable.name + "_" + cl.name;
     type := cl.type;
    };
    at.column += newCol;
    at.hasKey.column += newCol;
    sourceTableFKey.column += newCol;
   };

   -- Create a foreign key in the assoc table for the dest table
   var destTableFKey := object ForeignKey {
    name := destTable.name + "_fk";
    refersTo := destTable.hasKey;
   };
   at.hasFKey += destTableFKey;

   -- Create a column in the assoc table for each dest table id col
   destTable.hasKey.column->forEach(cl) {
    var newCol := object Column {
     name := destTable.name + "_" + cl.name;
     type := cl.type;
    };
    at.column += newCol;
    at.hasKey.column += newCol;
    destTableFKey.column += newCol;
   };
  };
}

query UmlTypeToDbtype(umlType:String) : String {
 if (umlType = "Integer") then {
  return "NUMBER";
 } endif;

 if (umlType = "String") then {
  return "VARCHAR";
 } endif;

 if (umlType = "Boolean") then {
  return "BOOLEAN";
 } endif;

 if (umlType = "Date") then {
  return "DATETIME";
 } endif;

 assert fatal (false) with log("Unable to translate UML type "
   + umlType + " to DB type");
 return null;
}

@varpoint {
 name := UsePartitioning,
 description := "Selects whether a class should be mapped onto
   a single table or a partition of tables",
 analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
 impact := {
  nfps::GlobalResponseTime(_),
  nfps::ResponseTime(_, $"self")
 }
}
mapping Class::UsePartitioning_VariationPoint()
  disjuncts
    Class::ClassToSingleTable_Variant,
    Class::ClassToPartition_Variant;

@variant {
 name := ClassToSingleTable,
 description := "Maps a class onto a single table."
}
mapping Class::ClassToSingleTable_Variant() {
 log("Mapping class " + self.name + " to single table");
```

```
 var mappingTable := self.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);
 mappingTable.partitionId := 0;
}

@variant {
 name := ClassToPartition,
 description := "Maps a class onto a partition.",
 excludes := RandomizedGeneration($"self.attribute")
}
mapping Class::ClassToPartition_Variant() {
 log("Mapping class " + self.name + " to single partition");

 -- Retrieve all subtables
 var subClasses := self.getAllSubClasses();
 var subTables := subClasses->resolveIn(Class::ClassToTableMarker)
   ->oclAsType(Table);

 -- Create a partition
 var mappingTable := self.resolveoneIn(Class::ClassToTableMarker)
   .oclAsType(Table);
 var schema := mappingTable.schema;

 -- Create a new partition
 var partition := object Partition {
  name := self.name;
  mappingTable.partitionId := 0;

  -- Attach table to the partition
  table += mappingTable;

  -- Attach all the subtables if they exist
  table += subTables;
 };
 schema.partition += partition;

 -- Invoke marker relationship
 self.map ClassToPartitionMarker(partition);

 -- At least two slices in the partition
 partition.size := 2;

 -- Check if more slices should be added
 self.map AddAnotherSlice_VariationPoint(partition.size + 1);
}

@varpoint {
 name := SlicesNumber,
 description := "Selects the number of slices to add in a partition",
 subjects := {#self},
 analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),
 impact := {
  nfps::GlobalResponseTime(_),
  nfps::ResponseTime(_, $"self")
 }
}
mapping Class::AddAnotherSlice_VariationPoint(in sliceId:Integer)
  disjuncts Class::AnotherSlice_Variant;

@variant {
 name := AnotherSlice,
 description := "Adds another slice to the partition."
}
mapping Class::AnotherSlice_Variant(in sliceId:Integer) {
 log("Adding slice " + sliceId.toString() + " for class " + self.name);

 var mappingPartition := self.resolveoneIn(Class::ClassToPartitionMarker)
   .oclAsType(Partition);
 mappingPartition.size := sliceId;

 self.map AddAnotherSlice_VariationPoint(sliceId + 1);
}

query Class::getAllSubClasses() : Set(Class) {
 return self.namespace.generalization->select(g | g.superType = self)
  ->collect(g | g.subType)->asSet();
}
```

A.2. The QVT-Relational Transformation

Listing A.2: The complete QVT-R transformation for the ORM case study.

```
transformation UmlToRdbms(uml:umlMM, rdbms:rdbmsMM)
{
 @nfpmodel[nfps => "Uml2Rdbms.nfps"]

 -- UML Keys
 key umlMM::Package {name};
 key umlMM::Classifier {name, namespace};
 key umlMM::Attribute {name, owner};
 key umlMM::PrimitiveDataType {name, namespace};
 key umlMM::Generalization {name, subType, superType};
 key umlMM::Association {name, source, destination};

 -- RDBMS Keys
 key rdbmsMM::Schema {name};
 key rdbmsMM::Partition {name,schema};
 key rdbmsMM::Table {name, partitionId, schema};
 key rdbmsMM::Column {name, owner};
 key rdbmsMM::Key {name, owner};
 key rdbmsMM::ForeignKey {name, owner, refersTo};

 -- This query maps Uml Data Types to RDBMS data types.
 query PrimitiveTypeToSqlType(primitiveTpe:String):String;
 query SqlTypeToPrimitiveType(sqlType:String):String;
 query GetMappingTable(class:umlMM::Class, schema:rdbmsMM::Schema)
   :rdbmsMM::Table;

 -- Maps a package to a schema.
 -- It is the starting top relation.
 top relation PackageToSchema
 {
  pn: String;

  checkonly domain uml p:Package { name = pn };

  enforce domain rdbms s:Schema {name = pn};
 }

 -- Maps a Persistent Top Class (a class with no super type) onto a table.
 top relation TopClassToTable
 {
  className : String;

  checkonly domain uml c:Class {
   namespace = p:Package{},
   name = className,
   kind = "Persistent"
  } {
   -- The class is a Top Class
   p.generalization->select(g|g.subType = c)->size() = 0
  };

  enforce domain rdbms t:Table {
   schema = s:Schema{},
   name = className,
   partitionId = 0
  };

  when { PackageToSchema(p,s); }

  where { ClassToSingleTable(c,t); }
 }

 -- Maps a Persistent Non-Top Class (a class with super type)
 -- onto a table, and decides which strategy to use for the mapping.
 top relation SubClassToTable {
  superClass : umlMM::Class;
  subTable : rdbmsMM::Table;

  checkonly domain uml c:Class {
   kind = "Persistent",
   namespace = p:Package{
    generalization = g:Generalization {
```

```
    superType = superClass,
    subType = c
   }
  }
 } {
  -- This is a Non-Top class
  c.namespace.generalization->select(g | g.subType = c)->size() <> 0
 };

 enforce domain rdbms superTable:Table { schema = s:Schema{} };

 when {
  PackageToSchema(p,s);

  -- We require the superclass to have been already mapped
  TopClassToTable(superClass,superTable)
    or SubClassToTable(superClass,superTable);
 }

 where {
  @varpoint {
   name := InheritanceStrategy,
   description := "Selects the strategy to map a generalization.",
   analyzer := examples.orm.qnanalyzer.QnAnalyzer(),
   impact := {
    nfps::GlobalResponseTime(_),
    nfps::ResponseTime(_, $"c")
   }
  }
  SubClassToParentTable(c, superTable)
    xor SubClassToTableAndAssoc(c, superTable);

  // We need these additional constraints to invalidate the checking
  // when using the bbox relation to intercept the variation point
  subTable = s.table->select(t | t.name = c.name)->any(true);
  subTable <> null or AllClassAttributesToSingleTable(c, superTable);
 }
}

@variant {
 name := UpwardCollapse,
 description := "Maps a subclass on the parent table."
}
relation SubClassToParentTable {

 checkonly domain uml c:Class { namespace = p:Package{}};

 enforce domain rdbms superTable:Table {
  schema = s:Schema{},
  column = dc:Column{
   name = "discriminator",
   type = "CHAR"
  }
 };

 when { PackageToSchema(p,s); }

 where { AllClassAttributesToSingleTable(c,superTable); }
}

@variant {
 name := AssociationCollapse,
 description := "Maps a subclass on its own table and on an association
   with the table onto which the parent was mapped."
}
relation SubClassToTableAndAssoc {
 cn, genName, fKeyName, pKeyName : String;

 superTable : rdbmsMM::Table;
 superTableKey : rdbmsMM::Key;

 checkonly domain uml c:Class {
  name = cn,
  namespace = p:Package{
   generalization = g:Generalization {
    name = genName,
    superType = superClass:Class{},
    subType = c
   }
  }
```

```
  };                                                                          153
                                                                              154
  enforce domain rdbms superTable:Table {                                     155
   schema = s:Schema {                                                        156
    table = subClassTable:Table{                                              157
     name = cn,                                                               158
     hasKey = genPKey:Key {                                                   159
      name = pKeyName                                                         160
     },                                                                       161
     hasFKey = genFKey:ForeignKey {                                           162
      name = fKeyName,                                                        163
      refersTo = superTableKey                                                164
     }                                                                        165
    }                                                                         166
   }                                                                          167
  } default_values {                                                          168
   pKeyName = cn + "_pk";                                                     169
   fKeyName = genName + "_fk";                                                170
  };                                                                          171
                                                                              172
  when { PackageToSchema(p,s); }                                              173
                                                                              174
  where {                                                                     175
   -- Map the identifier attributes of the parent class to the table         176
   -- mapping this class                                                     177
   IdAttributesToFKeyCols(superClass, genFKey);                               178
   IdAttributesToPKeyCols(superClass, genPKey);                               179
                                                                              180
   -- Enforces a mapping between the supertable primary key and              181
   -- this table foreign key                                                 182
   superTable = GetMappingTable(superClass,s);                                183
   superTableKey = superTable.hasKey;                                         184
                                                                              185
   -- Maps all the class attributes to columns onto this table               186
   AllClassAttributesToSingleTable(c,subClassTable);                          187
  }                                                                           188
}                                                                             189
                                                                              190
-- Maps a class onto a single table.                                          191
relation ClassToSingleTable {                                                 192
 cn : String;                                                                 193
 pkeyName : String;                                                           194
                                                                              195
 checkonly domain uml c:Class { name = cn };                                  196
                                                                              197
 enforce domain rdbms t:Table {                                               198
  name = cn,                                                                  199
  hasKey=k:Key {                                                              200
   name = pkeyName                                                            201
  }                                                                           202
 } default_values {                                                           203
  pkeyName = cn + "_pk";                                                      204
 };                                                                           205
                                                                              206
 where {                                                                      207
  IdAttributesToAutogenPKeyCols(c,k);                                         208
  AllClassAttributesToSingleTable(c,t);                                       209
                                                                              210
  -- Fix to disable triggering this relation when class mapped on partition   211
  t.partition = null or t.partition->size() = 0;                              212
 }                                                                            213
}                                                                             214
                                                                              215
-- Maps OneToN associations to foreign keys on the table onto which           216
-- the N cardinality class is mapped.                                         217
top relation OneToNAssocToForeignKey                                          218
{                                                                             219
 fkeyName : String;                                                           220
 assocName : String;                                                          221
                                                                              222
 s : rdbmsMM::Schema;                                                         223
 onePointTable, onePointTable_Dummy : rdbmsMM::Table;                         224
 nPointTable, nPointTable_Dummy : rdbmsMM::Table;                             225
 onePointTablePKey : rdbmsMM::Key;                                            226
                                                                              227
 checkonly domain uml a:Association {                                         228
  name = assocName,                                                           229
  owner = p:Package{},                                                        230
  source = sc:Class {                                                         231
   kind = "Persistent"                                                        232
```

```
  },                                                                          233
                                                                              234
  destination = dc:Class {                                                    235
   kind = "Persistent"                                                        236
  }                                                                           237
 } {                                                                          238
  -- The source endpoint has a non infinite cardinality                       239
  a.sourceCardinality <> -1                                                   240
 };                                                                           241
                                                                              242
 enforce domain rdbms fkey:ForeignKey {                                       243
  name = fkeyName,                                                            244
  owner = nPointTable,                                                        245
  refersTo = onePointTablePKey                                                246
 } default_values {                                                           247
  fkeyName = assocName + "_fk";                                               248
 };                                                                           249
                                                                              250
 when {                                                                       251
  PackageToSchema(p,s);                                                       252
                                                                              253
  -- The classes involved in the association have been already mapped         254
  TopClassToTable(sc,onePointTable_Dummy)                                     255
    or SubClassToTable(sc,onePointTable_Dummy);                               256
  TopClassToTable(dc,nPointTable_Dummy)                                       257
    or SubClassToTable(dc,nPointTable_Dummy);                                 258
                                                                              259
  nPointTable = GetMappingTable(dc,s);                                        260
  onePointTable = GetMappingTable(sc,s);                                      261
                                                                              262
  onePointTablePKey = onePointTable.hasKey;                                   263
 }                                                                            264
                                                                              265
 where { IdAttributesToFKeyCols(sc,fkey); }                                   266
}                                                                             267
                                                                              268
-- Maps NToOne associations to foreign keys on the table onto which           269
-- the N cardinality class is mapped.                                         270
top relation NToOneAssocToForeignKey                                          271
{                                                                             272
 fkeyName : String;                                                           273
 assocName : String;                                                          274
                                                                              275
 s : rdbmsMM::Schema;                                                         276
 onePointTable, onePointTable_Dummy : rdbmsMM::Table;                         277
 nPointTable, nPointTable_Dummy : rdbmsMM::Table;                             278
 onePointTablePKey : rdbmsMM::Key;                                            279
                                                                              280
 checkonly domain uml a:Association {                                         281
  name = assocName,                                                           282
  owner = p:Package{},                                                        283
  source = sc:Class {                                                         284
   kind = "Persistent"                                                        285
  },                                                                          286
                                                                              287
  destination = dc:Class {                                                    288
   kind = "Persistent"                                                        289
  }                                                                           290
 } {                                                                          291
  -- The destination endpoint has a non infinite cardinality                  292
  a.destCardinality <> -1                                                     293
 };                                                                           294
                                                                              295
 enforce domain rdbms                                                         296
 fkey:ForeignKey {                                                            297
  name = fkeyName,                                                            298
  owner = nPointTable,                                                        299
  refersTo = onePointTablePKey                                                300
 } default_values {                                                           301
  fkeyName = assocName + "_fk";                                               302
 };                                                                           303
                                                                              304
 when {                                                                       305
  PackageToSchema(p,s);                                                       306
                                                                              307
  -- The classes involved in the association have been already mapped         308
  TopClassToTable(sc,nPointTable_Dummy)                                       309
    or SubClassToTable(sc,nPointTable_Dummy);                                 310
  TopClassToTable(dc,onePointTable_Dummy)                                     311
    or SubClassToTable(dc,onePointTable_Dummy);                               312
```

```
    nPointTable = GetMappingTable(sc,s);
    onePointTable = GetMappingTable(dc,s);

    onePointTablePKey = onePointTable.hasKey;
   }

   where { IdAttributesToFKeyCols(dc,fkey); }
  }

  -- Maps NtoN associations
  -- Requires ClassToTable to hold for both the association endpoints
  -- Invokes pkeyColsToFKeyCols
  top relation NtoNAssocToSupportTable
  {
   assocName : String;
   sourceFKeyName, destFKeyName : String;
   pKeyName : String;

   rest : Set(rdbmsMM::ForeignKey);
   sourceTable, sourceTable_Dummy : rdbmsMM::Table;
   destTable, destTable_Dummy : rdbmsMM::Table;
   sourceKey, destKey : rdbmsMM::Key;

   checkonly domain uml a:Association {
    name = assocName,
    owner = p:Package{},

    -- Source class is persistent and has N cardinality
    source = sc:Class {
     kind = "Persistent"
    },
    sourceCardinality = -1,

    -- Destination class is persistent and has N cardinality
    destination = dc:Class {
     kind = "Persistent"
    },
    destCardinality = -1
   };

   enforce domain rdbms supportTable:Table {
    name = assocName,
    schema = s:Schema{},
    partitionId = 0,
    hasFKey = fkeys:Set(ForeignKey) {
     sourceFKey:ForeignKey{
      name = sourceFKeyName,
      refersTo = sourceKey
     },
     destFKey:ForeignKey{
      name = destFKeyName,
      refersTo = destKey
     }
     ++ rest
    },
    hasKey = pKey:Key{
     name = pKeyName
    }
   } default_values {
    rest = Set {};
    sourceFKeyName = assocName + "_" + sc.name + "_fk";
    destFKeyName = assocName + "_" + dc.name + "_fk";

    pKeyName = assocName + "_pk";
   };

   when {
    PackageToSchema(p,s);

    -- The classes involved in the association have been already mapped
    TopClassToTable(sc,sourceTable_Dummy)
      or SubClassToTable(sc,sourceTable_Dummy);
    TopClassToTable(dc,destTable_Dummy)
      or SubClassToTable(dc,destTable_Dummy);

    sourceTable = GetMappingTable(sc,s);
    destTable = GetMappingTable(dc,s);

    sourceKey = sourceTable.hasKey;
```

```
  destKey = destTable.hasKey;
 }

 where {
  -- Source class identifier attributes are mapped onto a foreign key
  IdAttributesToFKeyCols(sc,sourceFKey);

  -- Dest class identifier attributes are mapped onto a foreign key
  IdAttributesToFKeyCols(dc,destFKey);

  -- Add all columns to the primary key
  IdAttributesToPKeyCols(sc, pKey);
  IdAttributesToPKeyCols(dc, pKey);
 }
}

-- Maps the id attributes of a class onto primary key columns.
relation IdAttributesToPKeyCols {

 checkonly domain uml sourceClass:Class {};

 enforce domain rdbms pKey:Key {};

 where {
  IdAttributesToPKeyCols_Inner(sourceClass, pKey);

  -- Recurse only if this Class has not identifer attributes
  if (sourceClass.attribute->select(a | a.isId = true)->size() = 0 and
   sourceClass.namespace.generalization->exists(g | g.subType = sourceClass))
  then
   IdAttributesToPKeyCols(
   sourceClass.namespace.generalization->any(g | g.subType = sourceClass)
     .superType, pKey)
  else true
  endif;
 }
}

relation IdAttributesToPKeyCols_Inner {
 attrName, colName : String;

 checkonly domain uml sourceClass:Class {
  attribute = a:Attribute {
   name = attrName,
   isId = true
  }
 };

 enforce domain rdbms k:Key {
  column = cl:Column {
   name = colName,
   owner = k.owner
  }
 } default_values {
  colName = "fk_" + sourceClass.name + "_" + attrName;
 };

 where { AttributeToColumn(a, cl, "fk_" + sourceClass.name + "_"); }
}

-- Maps Identifier Attributes to Foreign Key Columns recursively.
relation IdAttributesToFKeyCols {
 checkonly domain uml sourceClass:Class { };

 enforce domain rdbms fKey:ForeignKey { };

 where {
  IdAttributesToFKeyCols_Inner(sourceClass,fKey);

  -- Recurse only if this Class has not identifier attributes
  if (sourceClass.attribute->select(a | a.isId = true)->size() = 0 and
   sourceClass.namespace.generalization->exists(g | g.subType = sourceClass))
  then
   IdAttributesToFKeyCols(
   sourceClass.namespace.generalization->any(g | g.subType = sourceClass)
     .superType, fKey)
  else true
  endif;
 }
}
```

```
                                                                                                    473
-- Maps Identifier Attributes to Foreign Key Columns                                                474
relation IdAttributesToFKeyCols_Inner {                                                             475
 attrName : String;                                                                                 476
 fkeyColName : String;                                                                              477
                                                                                                    478
 checkonly domain uml sourceClass:Class {                                                           479
  attribute = a:Attribute {                                                                         480
   name = attrName,                                                                                 481
   isId = true                                                                                      482
  }                                                                                                 483
 };                                                                                                 484
                                                                                                    485
 enforce domain rdbms fKey:ForeignKey {                                                             486
  column = cl:Column {                                                                              487
   name = fkeyColName,                                                                              488
   owner = fKey.owner                                                                               489
  }                                                                                                 490
 } default_values {                                                                                 491
  fkeyColName = "fk_" + sourceClass.name + "_" + attrName;                                          492
 };                                                                                                 493
                                                                                                    494
 where {                                                                                            495
  AttributeToColumn(a, cl, "fk_" + sourceClass.name + "_");                                         496
 }                                                                                                  497
}                                                                                                   498
                                                                                                    499
-- Maps the id attributes of a class onto primary key columns                                       500
-- with autogeneration.                                                                            501
relation IdAttributesToAutogenPKeyCols {                                                            502
 attrName : String;                                                                                 503
                                                                                                    504
 pkeyName : String;                                                                                 505
 pkeyColType : String;                                                                              506
                                                                                                    507
 checkonly domain uml c:Class {                                                                     508
  attribute = a:Attribute {                                                                         509
   name = attrName,                                                                                 510
   isId = true                                                                                      511
  }                                                                                                 512
 };                                                                                                 513
                                                                                                    514
 enforce domain rdbms k:Key {                                                                       515
  column = cl:Column {                                                                              516
   name = attrName,                                                                                 517
   owner = k.owner                                                                                  518
  }                                                                                                 519
 };                                                                                                 520
                                                                                                    521
 where {                                                                                            522
  AttributeToColumn(a, cl, "");                                                                     523
                                                                                                    524
  @varpoint {                                                                                       525
   name := AutogenerationStrategy,                                                                  526
   description := "Selects the strategy to auto generate values for id attributes",                 527
   analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),                                528
   impact := {                                                                                      529
    nfps::GlobalResponseTime(_),                                                                    530
    nfps::ResponseTime(_, $"c.owner")                                                               531
   }                                                                                                532
  }                                                                                                 533
  SingleIdAttributeToAutoGenSequenceCol(a,cl)                                                       534
    xor SingleIdAttributeToAutoGenSparseCol(a,cl);                                                  535
 }                                                                                                  536
}                                                                                                   537
                                                                                                    538
@variant {                                                                                          539
 name := SequentialGeneration,                                                                      540
 description := "A sequential generation strategy."                                                 541
}                                                                                                   542
relation SingleIdAttributeToAutoGenSequenceCol {                                                    543
 policy : String;                                                                                   544
                                                                                                    545
 checkonly domain uml a:Attribute {                                                                 546
 } {                                                                                                547
  a.owner.attribute->select(attr | attr.isId = true)->size() = 1                                    548
 };                                                                                                 549
                                                                                                    550
 enforce domain rdbms cl:Column {                                                                   551
  type = "NUMBER",                                                                                  552
```

```
  autoGenerate = policy                                                    553
 } {                                                                       554
  cl.owner.hasKey.column->size() = 1                                       555
 } default_values {                                                        556
  policy = "SEQUENCE";                                                     557
 };                                                                        558
                                                                           559
 where { policy = "SEQUENCE"; }                                            560
}                                                                          561
                                                                           562
@variant {                                                                 563
 name := RandomizedGeneration,                                             564
 description := "A randomized generation strategy."                        565
}                                                                          566
relation SingleIdAttributeToAutoGenSparseCol {                             567
 policy : String;                                                          568
                                                                           569
 checkonly domain uml a:Attribute {                                        570
 } {                                                                       571
  a.owner.attribute->select(attr | attr.isId = true)->size() = 1           572
 };                                                                        573
                                                                           574
 enforce domain rdbms cl:Column {                                          575
  type = "NUMBER",                                                         576
  autoGenerate = policy                                                    577
 } {                                                                       578
  cl.owner.hasKey.column->size() = 1                                       579
 } default_values {                                                        580
  policy = "SPARSE";                                                       581
 };                                                                        582
                                                                           583
 where { policy = "SPARSE"; }                                              584
}                                                                          585
                                                                           586
-- Map all the non id  attributes of a class onto columns of a table.     587
relation AllClassAttributesToSingleTable {                                 588
 attrName : String;                                                        589
 markId : Boolean;                                                         590
                                                                           591
 checkonly domain uml c:Class {                                            592
  attribute = a:Attribute {                                                593
   name = attrName,                                                        594
   isId = markId                                                           595
  }                                                                        596
 };                                                                        597
                                                                           598
 enforce domain rdbms t:Table {                                            599
  column = cl:Column {                                                     600
   name = attrName                                                         601
  }                                                                        602
 };                                                                        603
                                                                           604
 where {                                                                   605
  AttributeToColumn(a, cl, "");                                            606
  markId = (cl.hasKey->size() <> 0);                                       607
 }                                                                         608
}                                                                          609
                                                                           610
-- Maps an attribute to a column.                                          611
relation AttributeToColumn {                                               612
 attrName : String;                                                        613
 attrTypeName, colTypeName : String;                                       614
 namePrefix : String;                                                      615
 prefixedName : String;                                                    616
                                                                           617
 checkonly domain uml a:Attribute {                                        618
  name = attrName,                                                         619
  type=p:PrimitiveDataType {                                               620
   name=attrTypeName,                                                      621
   namespace = a.owner.namespace                                           622
  }                                                                        623
 } default_values {                                                        624
  attrTypeName = SqlTypeToPrimitiveType(colTypeName);                      625
 };                                                                        626
                                                                           627
 enforce domain rdbms c:Column {                                           628
  name = prefixedName,                                                     629
  type = colTypeName                                                       630
 } default_values {                                                        631
  prefixedName = namePrefix + attrName;                                    632
```

```
  };                                                                              633
                                                                                  634
  primitive domain prefix : String;                                               635
                                                                                  636
  when { namePrefix = prefix; }                                                   637
                                                                                  638
  where { colTypeName = PrimitiveTypeToSqlType(attrTypeName); }                   639
 }                                                                                640
                                                                                  641
 -- Maps primitive uml types to sql types.                                        642
 query PrimitiveTypeToSqlType(primitiveType:String):String                        643
 {                                                                                644
  if (primitiveType="Integer")                                                    645
  then "NUMBER"                                                                   646
  else                                                                            647
   if (primitiveType="Boolean")                                                   648
   then "BOOLEAN"                                                                 649
   else                                                                           650
    if (primitiveType="String")                                                   651
    then "VARCHAR"                                                                652
    else                                                                          653
     if (primitiveType="Date")                                                    654
     then "DATETIME"                                                              655
     else "UNK"                                                                   656
     endif                                                                        657
    endif                                                                         658
   endif                                                                          659
  endif                                                                           660
 }                                                                                661
                                                                                  662
 -- Maps sql types to uml primitive types.                                        663
 query SqlTypeToPrimitiveType(sqlType:String):String                              664
 {                                                                                665
  if (sqlType="NUMBER")                                                           666
  then "Integer"                                                                  667
  else                                                                            668
   if (sqlType="BOOLEAN")                                                         669
   then "Boolean"                                                                 670
   else                                                                           671
    if (sqlType="VARCHAR")                                                        672
    then "String"                                                                 673
    else                                                                          674
     if (sqlType="DATETIME")                                                      675
     then "Date"                                                                  676
     else "UNK"                                                                   677
     endif                                                                        678
    endif                                                                         679
   endif                                                                          680
  endif                                                                           681
 }                                                                                682
                                                                                  683
 query GetMappingTable(class:umlMM::Class, schema:rdbmsMM::Schema)                684
   :rdbmsMM::Table                                                                685
 {                                                                                686
  if (schema.table->select(t | t.name = class.name)->size() <> 0)                 687
  then schema.table->select(t | t.name = class.name)->any(true)                   688
  else GetMappingTable(                                                           689
   class.namespace.generalization->select(g | g.subType = class)->any(true)       690
     .superType, schema)                                                          691
  endif                                                                           692
 }                                                                                693
}                                                                                 694
```

B. The Component Allocation Transformation

This appendix contains the complete listing of the transformation with QVT-Operational for the component allocation case study.

Listing B.1: The complete QVT-O transformation for the component allocation case study.

```
modeltype components "strict" uses components(
  "http://qvtr2.org/examples/allocation/components");
modeltype hardware "strict" uses hardware(
  "http://qvtr2.org/examples/allocation/hardware");
modeltype allocation "strict" uses allocation(
  "http://qvtr2.org/examples/allocation");

@nfpmodel[nfps => "allocation.nfps"]

@keydefs{
  components::ComponentModel(#name),
  components::Module(#name, #owner),
  components::Component(#name, #owner),
  components::Operation(#name, #owner),

  hardware::HardwareModel(#name),
  hardware::LinkableElement(#name),
  hardware::Cluster(#name),
  hardware::Resource(#name),
  hardware::Link(#name),
  hardware::P2PLink(#name),
  hardware::Bus(#name),

  allocation::AllocationModel(#name),
  allocation::Allocation(#name)
}

transformation Allocation(in cm:components, in hm:hardware,
  out am:allocation);
main() {
  -- Check models are well formed
  assert fatal (cm.rootObjects()[ComponentModel]->size() = 1)
    with log ("Malformed component model");
  assert fatal (hm.rootObjects()[HardwareModel]->size() = 1)
    with log ("Malformed hardware model");
  assert fatal (am.rootObjects()[AllocationModel]->size() <= 1)
    with log ("Malformed allocation model");

  -- Retrieve the necessary resources
  var componentModel:ComponentModel := cm.rootObjects()[ComponentModel]
    ->any(true);
  var hardwareModel:HardwareModel := hm.rootObjects()[HardwareModel]
    ->any(true);

  var allComponents := componentModel.modules.components;
  var allHosts := hardwareModel.resources;

  var allocationModel := am.map getAllocationModel(componentModel,
    hardwareModel);

  allComponents->map allocate(allHosts);

  assert fatal (checkConnectivity(allComponents))
    with log ("Connectivity constraint violated");
}
```

```
                                                                                    56
-- Gets or creates the allocation model                                             57
mapping allocation::getAllocationModel(in componentModel:ComponentModel,            58
  in hardwareModel:HardwareModel) : AllocationModel                                 59
  disjuncts                                                                         60
    allocation::getExistingAllocationModel,                                         61
    allocation::getNewAllocationModel;                                              62
                                                                                    63
-- Gets an existing allocation model                                                64
mapping allocation::getExistingAllocationModel(                                     65
  in componentModel:ComponentModel, in hardwareModel:HardwareModel)                 66
  : AllocationModel                                                                 67
when {                                                                              68
  self.rootObjects()[AllocationModel]->size() = 1                                   69
}{                                                                                  70
init { result := am.rootObjects()[AllocationModel]->any(AllocationModel); }         71
}                                                                                   72
                                                                                    73
-- Creates a new allocation model                                                   74
mapping allocation::getNewAllocationModel(                                          75
  in componentModel:ComponentModel, in hardwareModel:HardwareModel)                 76
  : AllocationModel                                                                 77
{                                                                                   78
  result.name := componentModel.name + "_" +                                        79
    hardwareModel.name + "_allocation";                                             80
  result.hardwareModel := hardwareModel;                                            81
  result.componentsModel := componentModel;                                         82
}                                                                                   83
                                                                                    84
mapping Component::allocate(in availableHosts:OrderedSet(Resource))                 85
  : Sequence(Allocation)                                                            86
when {                                                                              87
  not self.isAlreadyAllocated();                                                    88
} {                                                                                 89
  availableHosts->forEach(h) {                                                      90
    if (availableHosts->indexOf(h) = availableHosts->size()) then {                 91
      log ("Default allocation");                                                   92
      -- If this is the last available host, we can only map it here              93
      self.map defaultAllocate(h);                                                  94
      break;                                                                        95
    } endif;                                                                        96
                                                                                    97
    -- Otherwise decide to allocate or not                                          98
    self.map allocateOn_VariationPoint(h);                                          99
                                                                                    100
    -- Exit loop if component has been allocated                                    101
    if (self.isAlreadyAllocated()) then {                                           102
      break;                                                                        103
    } endif;                                                                        104
  };                                                                                105
}                                                                                   106
                                                                                    107
@varpoint {                                                                         108
  name := ComponentAllocation,                                                      109
  description := "Decides about the allocation of a component onto                  110
    a specific resource.",                                                          111
  analyzer := examples.allocation.Analyzer(),                                       112
  impact := {                                                                       113
    nfps::Schedulability($"self"),                                                  114
    nfps::GlobalSchedulability(),                                                   115
    nfps::Clustering($"self.owner"),                                                116
    nfps::GlobalClustering(),                                                       117
    nfps::GlobalReliability(),                                                      118
    nfps::ComponentReliability($"self"),                                            119
    nfps::OperationReliability($"self", $"self.operations"),                        120
    nfps::Cost(),                                                                   121
    nfps::LinkCost(),                                                               122
    nfps::ResourceCost()                                                            123
  }                                                                                 124
}                                                                                   125
mapping Component::allocateOn_VariationPoint(in host:Resource)                      126
  disjuncts                                                                         127
    Component::dontAllocateOn_Variant,                                              128
    Component::allocateOn_Variant;                                                  129
                                                                                    130
@variant {                                                                          131
  name := Allocate,                                                                 132
  description := "Allocates the component on the hardware resource.",               133
  requires := Allocate($"self.module.components", $"host")                          134
}                                                                                   135
```

```
mapping Component::allocateOn_Variant(in host:Resource) {
  log ("Allocating component " + self.name + "on host " + host.name);
  getAllocationModel().allocations += object allocation::Allocation {
    name := self.name + " on " + host.name;
    component := self;
    resource := host;
  };
}

@variant {
  name := DontAllocate,
  description := "Does not allocate the component on the hardware resource."
}
mapping Component::dontAllocateOn_Variant(in host:Resource) {
  -- Just do nothing
  log ("Skipping allocation of component " + self.name +
    "on host " + host.name);
}

mapping Component::defaultAllocate(in host:Resource) {
  log ("Allocating component " + self.name + "on host " + host.name);
  getAllocationModel().allocations += object allocation::Allocation {
    name := self.name + " on " + host.name;
    component := self;
    resource := host;
  };
}

-- Checks if a component has been already allocated
query Component::isAlreadyAllocated() : Boolean {
  return getAllocationModel().allocations->exists(a | a.component = self);
}

-- Gets the allocation model
query getAllocationModel() : AllocationModel {
  assert fatal (am.rootObjects()[AllocationModel]->size() = 1)
    with log ("Unable to find allocation model");
  return am.rootObjects()[AllocationModel]->any(true)
    .oclAsType(AllocationModel);
}

query checkConnectivity(in comps:Sequence(Component)) : Boolean {
  var allocationModel := getAllocationModel();

  -- Compute reachability map
  var hardwareModel:HardwareModel := hm.rootObjects()[HardwareModel]
    ->any(true);
  var reachabilityMap:Dict(Resource, Set(Resource));
  hardwareModel.resources->forEach(r) {
    reachabilityMap->put(r, computeReachableResources(r,
      reachabilityMap));
  };

  comps->forEach(c) {
    var compAllocations := allocationModel.allocations->any(a |
      a.component = c);
    var compResource := compAllocations.resource;

    var usedComponents := c.operations._uses.owner;
    var usedComponentsAllocations := allocationModel.allocations
      ->select(a | usedComponents->includes(a.component));
    var usedResources := usedComponentsAllocations.resource->asSet();

    if (not reachabilityMap->get(compResource)->includesAll(
      usedResources)) then {
      return false;
    } endif;
  };

  return true;
}

query computeReachableResources(
  in resource:Resource,
  in reachabilityMap:Dict(Resource, Set(Resource))) :
  Set(Resource) {

  -- The list of visited resources
  var visited:Set(Resource);
```

```
  -- The list of elements to visit during the next                    216
  -- iteration of the loop.                                           217
  var toVisit:Set(LinkableElement);                                   218
  toVisit += resource;                                                219
                                                                      220
  while (not toVisit->isEmpty()) {                                    221
    -- The list of discovered elements during this iteration          222
    var discovered:Set(LinkableElement);                              223
                                                                      224
    toVisit->forEach(le) {                                            225
      -- Chech if we are visiting a cluster                           226
      if (le.oclIsKindOf(Cluster)) then {                             227
        -- This means that we are able to connect to everything inside 228
        -- the cluster.                                               229
        var currentCluster := le.oclAsType(Cluster);                  230
                                                                      231
        -- Add to the discovered set all the cluster resources        232
        -- not already visited                                        233
        discovered += currentCluster.resources->select(r |            234
          visited->excludes(r));                                      235
        continue;                                                     236
      } endif;                                                        237
                                                                      238
      -- We end up here only if the current le is a resource          239
      var currentResource := le.oclAsType(Resource);                  240
                                                                      241
                                                                      242
      -- If we already computed the reachability set for the current  243
      -- resource just put the results in the visited set             244
      if (reachabilityMap->hasKey(currentResource)) then {            245
        visited += reachabilityMap->get(currentResource);             246
                                                                      247
        -- We do not need to add the current resource to the toVisit set 248
        continue;                                                     249
      } endif;                                                        250
                                                                      251
      -- If we already visited the current resource (i.e., there is a loop) 252
      -- just skip it                                                 253
      if (visited->includes(currentResource)) then {                  254
        continue;                                                     255
      } endif;                                                        256
                                                                      257
      -- We end up here if this is the first time we visit the resource 258
      -- and there is no reachability set computed                    259
                                                                      260
      -- Add the resource to the visited set                          261
      visited += currentResource;                                     262
                                                                      263
      -- Update the discovered target by following the links from the 264
      -- current resource                                             265
      discovered += le.outgoingLinks.destination;                     266
      discovered += le.buses.connectedHosts;                          267
    };                                                                268
                                                                      269
    -- The new toVisit list is the discovered list                    270
    -- Update to visit list                                           271
    toVisit := discovered;                                            272
  };                                                                  273
                                                                      274
  return visited;                                                     275
}                                                                     276
```

Bibliography

[1] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, and Vincenzo Grassi. Klapersuite: An integrated model-driven environment for reliability and performance analysis of component-based systems. In *TOOLS*, volume 6705 of *LNCS*, pages 99–114, 2011.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *WOSP*, pages 54–65. ACM, 2007.

[3] Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (puma). In *WOSP*, pages 1–12. ACM, 2005.

[4] T. Bures, J. Carlson, J.I. Crnkovic, S. Sentilles, and A. Vulgarakis. Procom - the progress component model reference manual, version 1.0. Technical Report MHD-MRTC-230/2008-1-SE, Malardalen University, June 2008.

[5] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE TSE*, 30(5):295–310, 2004.

[6] *Proceedings of the International Workshop on Software and Performance (WOSP)*. ACM.

[7] Trevor Parsons. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In *DSM*. ACM, 2005.

[8] Jing Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP*. ACM, 2008.

[9] Vittorio Cortellessa, Anne Martens, Ralf Reussner, and Catia Trubiani. A process to effectively identify "guilty" performance antipatterns. In *FASE*, 2010.

[10] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *WOSP/SIPEW*, 2010.

[11] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *GECCO*. ACM, 2005.

[12] Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *MOMPES*. IEEE, 2009.

[13] Tripti Saxena and Gabor Karsai. Mde-based approach for generalizing design space exploration. In *MoDELS*. Springer, 2010.

[14] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: Towards generic automation for mda. In *EMSOFT*. ACM, 2010.

[15] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*. Springer, 2003.

[16] J Merilinna. *A Tool for Quality-Driven Architecture Model Transformation*. PhD thesis, VVT Technical Research Centre of Finland, Vuorimiehentie, Finland, 2005.

[17] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, Unversity of Twente, Twente, Netherlands, 2005.

[18] Emilio Insfrán, Javier Gonzalez-Huerta, and Silvia Abrahão. Design guidelines for the development of quality-driven model transformations. In *MoDELS*, volume 6395 of *LNCS*, pages 288–302. Springer, 2010.

[19] Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. A quality driven extension to the qvt-relations transformation language. *CSRD, Springer*, 2011.

[20] Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. Towards quality driven exploration of model transformation spaces. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 2–16, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, 39(2):59 – 66, feb. 2006.

[22] N. Rochette. Introduction to executable uml, lecture held at mdd4dres spring school, 2009.

[23] Object Management Group (OMG). Unified modeling language (uml) 2.3, superstructure, May 2010.

[24] Soley, R. and the OMG Staff. Model-driven architecture. *OMG Document*, November 2000.

[25] Object Management Group (OMG). Common object request broker architecture (corba/iiop), January 2008.

[26] Oracle. Java enterprise edition (javaee). `http://www.oracle.com/technetwork/java/javaee/overview/index.html`. Last accessed July 2011.

[27] Jean Bézivin. Model driven engineering: An emerging technical space. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 36–64. Springer, 2006.

[28] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In *UML*, volume 2863, pages 175–189, 2003.

[29] Stuart Kent. Model driven engineering. In *IFM*, volume 2335, pages 286–298, 2002.

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

[31] Jean-Marie Favre and Tam Nguyen. Towards a megamodel to model software evolution through transformations. *Electr. Notes Theor. Comput. Sci.*, 127(3):59–74, 2005.

[32] Object Management Groud (OMG). MetaObject Facility (MOF). `http://www.omg.org/spec/MOFFOL/2.0/`.

[33] The Eclipse Foundation. Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`.

[34] The Eclipse Foundation. Eclipse. `http://www.eclipse.org`.

[35] Microsoft Corporation. Visual Studio Visualization and Modeling SDK (VSVMSDK). `http://archive.msdn.microsoft.com/vsvmsdk`.

[36] Itemis a.g. Xtext. `http://www.eclipse.org/Xtext/`.

[37] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20:42–45, September 2003.

[38] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? transformation models! In *MoDELS*, volume 4199, pages 440–453, 2006.

[39] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model synchronisation: Definitions for round-trip engineering. In *ICMT*, volume 5063 of *LNCS*, pages 31–45, 2008.

[40] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[41] Object Management Group (OMG). Mof qvt specification 1.0, April 2008.

[42] P. A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J. M. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations in Practice Workshop at MoDELS 2005*, 2005.

[43] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.

[44] Andy Schürr. Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 903 of *LNCS*, pages 151–163. Springer, June 1994.

[45] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Model Transformations in Practice Workshop at MoDELS 2005*, number 3844 in LNCS. Springer, 2005.

[46] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[47] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. In *First International Workshop on Metamodeling for MDA*, 2003.

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

[49] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Generative Programming and Component Engineering (GPCE) Workshop*, 2004.

[50] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18 –41, jul 1993.

[51] Airfrance flight 447 accident. `http://en.wikipedia.org/wiki/Air_France_Flight_447`. Last Accessed July 2011.

[52] E.D. Lazowska, J. Zahorjan, G.S. Graham, and Sevcik K.C. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall, 1984.

[53] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[54] R.C. Cheung. A user-oriented software reliability model. In *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 565 – 570, 1978.

[55] Anne Immonen and Eila Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7:49–65, 2008. 10.1007/s10270-006-0040-x.

[56] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 3920:441–444, 2006.

[57] M. Kwiatkowska, G. Norman, and D. Parker. Prism 2.0: a tool for probabilistic model checking. *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 322–323, 2004.

[58] C. Baier, J.P. Katoen, et al. Principles of model checking. 2008.

[59] K. Goseva and K.S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Perform. Eval.*, 45:179–204, July 2001.

[60] Object Management Group (OMG). System modeling language 1.2, June 2010.

[61] Connie U. Smith and Lloyd G. Williams. *Performance and Scalability of Distributed Software Architectures: an SPE Approach.* Addison Wesley, 2002.

[62] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, 2010.

[63] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) 1.1.

[64] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[65] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, March 1999.

[66] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[67] Connie U. Smith and Lloyd G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software.* Addison Wesley, 2002.

[68] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns; common performance problems and their solutions. In *Int. CMG Conference*, pages 797–806, 2001.

[69] Connie U. Smith and Lloyd G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. CMG Conference*, pages 667–674, 2002.

[70] Connie U. Smith and Lloyd G. Williams. More new software antipatterns: Even more ways to shoot yourself in the foot. In *Int. CMG Conference*, pages 717–725, 2003.

[71] J. D. McGregor, F. Bachmann, L. Bass, P. Bianco, and M. Klein. Using arche in the classroom: One experience. Technical Report SEI-2007-TN-001, CMU, 2007.

[72] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[73] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley interscience series in discrete mathematics and optimization. Wiley, 1988.

[74] Lars Grunske, Peter A. Lindsay, Egor Bondarev, Yiannis Papadopoulos, and David Parker 0002. An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In *WADS*, volume 4615, pages 188–209, 2006.

[75] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In *WOSP*, pages 153–163, 2007.

[76] The ROBOCOP Component Model. `http://www.hitech-projects.com/euprojects/robocop/`.

[77] Amogh Kavimandan and Aniruddha S. Gokhale. Applying model transformations to optimizing real-time qos configurations in dre systems. In *QoSA*, volume 5581, pages 18–35, 2009.

[78] Brandon Eames, Sandeep Neema, and Rohit Saraswat. Desertfd: a finite-domain constraint based tool forÂădesign space exploration. *Design Automation for Embedded Systems*, 14:43–74, 2010. 10.1007/s10617-009-9049-z.

[79] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[80] Checkstyle. `http://checkstyle.sourceforge.net`.

[81] Sonar. `http://www.sonarsource.org`.

[82] Microsoft Corporation. Fxcop. `http://blogs.msdn.com/b/codeanalysis/`.

[83] Microsoft Corporation. Visual studio. `http://www.microsoft.com/visualstudio`.

[84] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[85] The scala programming language. `http://www.scala-lang.org/`.

[86] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 602–616, New York, NY, USA, 2006. ACM.

[87] Brian Henderson-Sellers. Uml - the good, the bad or the ugly? perspectives from a panel of experts. *Software and Systems Modeling*, 4:4–13, 2005. 10.1007/s10270-004-0076-8.

[88] Trevor Parsons and John Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 2008.

[89] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[90] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* ACM Press, 1997.

[91] Marten Sijtema. Managing variability in model transformations for model-driven product lines. Master's thesis, University of Twente, Enschede, The Netherlands, 2010.

[92] D. Alur, J. Crupi, and D. Malks. *Core J2EE patterns: best practices and design strategies.* Sun Microsystems Press, 2003.

[93] SUN Microsystems. Java pet store 2.0 reference application - blueprints. `https://blueprints.dev.java.net/petstore/`, Last Accessed 2010.

[94] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.

[95] Lars Grunske, Leif Geiger, and Michael Lawley. A graphical specification of model transformations with triple graph grammars. In *Model Driven Architecture, Foundations and Applications*, volume 3748 of *LNCS*, pages 284–298. Springer, 2005.

[96] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12:645–672, 2000.

[97] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.

[98] Rasha Osman, Irfan Awan, and Michael E. Woodward. Application of queueing network models in the performance evaluation of database designs. *Electron. Notes Theor. Comput. Sci.*, 232:101–124, March 2009.

[99] MetaCase. MetaEdit+. `http://www.metacase.com/`.

[100] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[101] IKV++ technologies ag. Medini-qvt. `http://projects.ikv.de/qvt`.

[102] Tata Research Inc. Modelmorf. `www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`.

[103] Object Management Group (OMG). Object constraint language (ocl), February 2010.

[104] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[105] T. H. Cormen, C. E. Leiserson, R. L Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[106] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[107] IBM. Java emitter templates. `http://www.eclipse.org/modeling/m2t/?project=jet`.

[108] The Eclipse Foundation. Model 2 Model (M2M) Initiative. `http://www.eclipse.org/m2m/`. Last accessed July 2011.

[109] S. Q. Lau and K. Czarnecki. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, Canada, 2006.

[110] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[111] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Logic Programming and Nonmonotonic Reasoning*, volume 2173, pages 434–438, 2001.

[112] Kousha Etessami and Mihalis Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. In *STACS*, pages 340–352, 2005.