# A Query Structured Model Transformation Approach

# A QUERY STRUCTURED MODEL TRANSFORMATION APPROACH

BY
HAMID MOHAMMAD GHOLIZADEH, M.Sc.

a thesis
submitted to the department of Computing and Software Engineering
and the school of graduate studies
of mcmaster university
in partial fulfilment of the requirements
for the degree of
Doctor of Philosophy

Doctor of Philosophy (2017)
(Software Engineering)

McMaster University
Hamilton, Ontario, Canada

| | |
|---|---|
| TITLE: | A Query Structured Model Transformation Approach |
| AUTHOR: | Hamid Mohammad Gholizadeh<br>M.Sc. in Information Technology (Software Design and Development) |
| SUPERVISORS: | Dr. Tom Maibaum, Dr. Zinovy Diskin |
| NUMBER OF PAGES: | xii, 151 |

*To my parents and wife*

# Abstract

Model Driven Engineering (MDE) has gained a considerable attention in the software engineering domain in the past decade. MDE proposes shifting the focus of the engineers from concrete artifacts (e.g., code) to more abstract structures (i.e., models). Such a change allows using the human intelligence more efficiently in engineering software products. Model Transformation (MT) is one of the key operations in MDE and plays a critical role in its successful application. The current MT approaches, however, usually miss either one or both of the two essential features: 1) declarativity in the sense that the MT definitions should be expressed at a sufficiently high level of abstraction, and 2) formality in the sense that the approaches should be based on precise underlying semantics. These two features are both critical in effectively managing the complexity of a network of interrelated models in an MDE process. This thesis tackles these shortcomings by promoting a declarative MT approach that is built on mathematical foundations. The approach is called Query Structured Transformation (QueST) as it proposes a structured orchestration of diagrammatic queries in the MT definitions. The aim of the thesis is to make the QueST approach –that is based on formal foundations– accessible to the MDE community. This thesis first motivates the necessity of having declarative formal approaches by studying the variety of model synchronization scenarios in the networks of interrelated models. Then, it defines a diagrammatic query framework (DQF) that formulates the syntax and the semantics of the QueST collection-level diagrammatic operations. By a detailed comparison of the QueST approach and three rule-based MT approaches (ETL, ATL, and QVT-R), the thesis shows the way QueST contributes to the development of the following aspects of MT definitions: declarativity, modularity, incrementality, and logical analysis of MT definitions.

# Acknowledgements

I would like to first thank my supervisors Dr. Tom Maibaum and Dr. Zinovy Diskin, who guided me during my Ph.D. studies and helped me in various ways with my research work. I also should thank the supervisory committee members Dr. Wolfram Kahl, and Dr. Jacques Carette for their reviews and comments. I also need to thank Dr. Ali Safilian and other office mates for their social and scientific supports, since most of my time spent with them in the office during my studies at McMaster University. Further, I should thank the people in the NECSIS[1] project including the researchers at McMaster University and the colleagues from other universities that all helped to shape the content of this thesis by providing feedback during talks and presentations in workshops and conferences. I also should acknowledge some anonymous reviewers commenting on the published papers from this thesis that contributed to the clarity of this thesis content.

In the end, I am greatly thankful to my parents who always generously offered their love and support and my wife whose patience and motivation let me keep completing this thesis work.

[1]Network on Engineering Complex Software Intensive Systems for Automotive Systems.

# Notations and Abbreviations

| | |
|---|---|
| **DPF** | Diagrammatic Predicate Framework |
| **DQF** | Diagrammatic Query Framework |
| **FOL** | First-Order Logic |
| **GT** | Graph Transformation |
| **MDA** | Model Driven Architecture |
| **MDE** | Model Driven Engineering |
| **MT** | Model Transformation |
| **NIM** | Network of Interrelated Models |
| **OCL** | Object Constraint Language |
| **PB** | Pullback |
| **PO** | Pushout |
| **QueST** | Query Structured Transformation |
| **UML** | Unified Modeling Language |

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Model Driven Engineering (MDE)

From the early days of computing, researchers and practitioners have attempted to create further abstractions over the low level programming structures. In this direction, Model Driven Engineering (MDE) proposes to even move further and shift the focus of engineers from code to models (Schmidt, 2006; Stahl and Völter, 2006; Kleppe *et al.*, 2003). Model Transformation (MT) –defined as a translation of one model to another model– is a key operation in MDE that plays a critical role in its successful application (Sendall and Kozaczynski, 2003; Zhang *et al.*, 2005). As MT is developed and motivated within the context of MDE, we begin this chapter by providing a brief history of MDE, its evolution, and its challenges. The subject of the current thesis arises from the challenges that MDE faces in terms of engineering model transformations.

### 1.1.1 MDE History

Since the early days of software engineering, the complexities of software systems kept increasing. To manage this complexity, a variety of techniques have been suggested, among which was increasing the level of abstraction. In particular, engineering software systems based on models rather than concrete code is a technique that is proposed by MDE and is shown to be promising (Selic, 2003; France and Rumpe, 2007; Schmidt, 2006; Stahl *et al.*, 2006; Beydeda *et al.*, 2005) at dealing with complex software system developments. In an MDE process, the focus of a software engineer shifts from the concrete level of implementation (code) to a more abstract level of specification (model), and the development task ideally becomes more about modelling and less about coding.

MDE has gained further attention in industry and the research community after Model-Driven Architecture (MDA) first version (Miller *et al.*, 2003) was released by OMG (Object Management Group) in 2000. MDA suggests the following ideal methodology in software development as an MDE process: platform-independent models describing a software system at a high-level of abstraction are transformed stepwise to platform-dependent models, from which executable source code is automatically generated. The generated code can be discarded anytime, whereas models are the primary artifacts to be maintained. Fig. 1.1(a) illustrates this pipeline-like flow of transformations: models are always transformed unidirectionally from a higher to a lower level of abstraction.

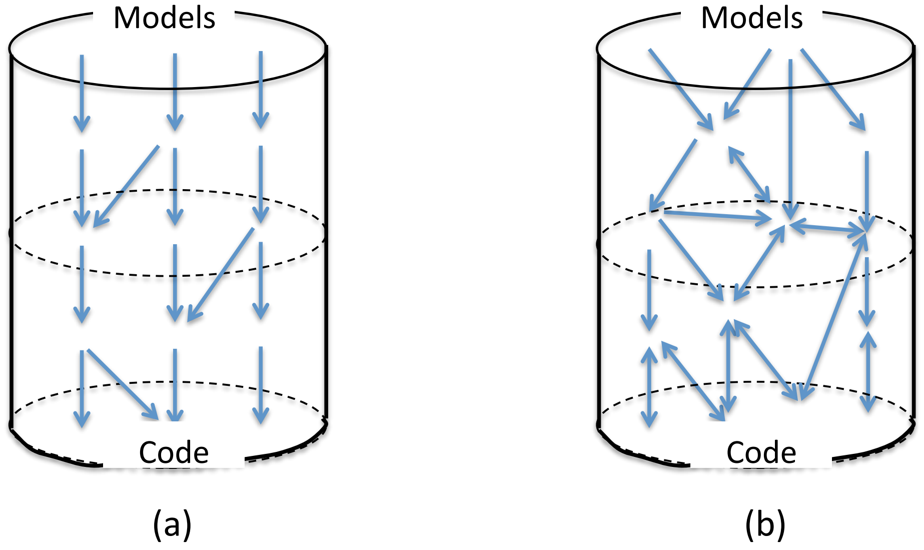


Figure 1.1: MDE approaches: (a) suggested by the MDA document (b) practical scenario that usually happens.

### 1.1.2 MDE Evolution

After the MDA proposal, it was soon revealed, however, that the suggested unidirectional pipeline structure illustrated in Fig. 1.1(a) is very difficult to achieve in practice, because of the following two reasons:

- Manual changes to the generated code (or lower level models) were often unavoidable in practice. Such changes make the models at the higher levels of abstraction obsolete if the changes are not propagated back to the higher level models. Thus, the arrows in Fig. 1.1(a) need to be bi-directional to represent vertically bi-directional transformations or *round-tripping*.

- Multiple high-level models often describe different aspects of a system in practice, and they have overlapping information; for example, the class diagrams and the sequence diagrams in UML (Arlow and Neustadt, 2005) each specify different (i.e., structural vs. behavioural) aspects of a software application; however, they share some common information (e.g., class names, and their corresponding method signatures). If one of these models at the same level of abstraction is changed, it is expected that the changes are propagated horizontally to the other models in the pipeline. From another perspective, models which encompass all aspects of a system may be too unwieldy and difficult to deal with in practice; this forces the modeller to capture different aspects of a system employing different (types of) models that might share information. Thus, bidirectional synchronization is not only needed vertically between the abstraction layers but also horizontally within the abstraction layers.

The above two points transform the MDA pipe in Fig. 1.1(a) into a more realistic view in Fig. 1.1(b) which sketches a *network of interrelated models* (NIM).

### 1.1.3 MDE Challenges

There exists a number of aspects we need to take into account when managing a network of interrelated model. In transformations between the models, we need to first separate the concerns and distinguish, for example, between the alignment, and synchronization procedure (i.e., update propagation). Alignment is a heuristic process of matching the elements in different models and often requires input from the user; however, update propagation can be treated as an algebraic operation amenable to full automation after an update policy is established (see Diskin *et al.* (2011a)). Another aspect is the multi-directionality of model synchronizations that means changes between models are propagated in all directions in a mutually consistent way. Its implementation via separate but mutually compatible procedures would require a proof of compatibility, and would be very difficult to maintain as each direction of change propagation is itself a complex model transformation. For the case of two models, a common solution nowadays is to specify a consistency relation between the models and leave the update propagation procedures to be inferred from this specification, so that they are always consistent by construction. Such an approach is commonly referred to as bidirectional transformation or *bx*. We are not aware of any implementation of the multi-directional case.

Specification of such complex synchronization procedures as discussed above is not well understood, whereas clear specification is crucial for synchronization tool builders and users; otherwise the tool builders would not have a sound foundation to characterize the behaviour of the tool, and tool users would have no trust in automatic synchronization carried out by such tools. Severe problems of adopting the industrial standard QVT-R (which treats the binary synchronization case) is a typical example: despite the early availability of QVT-R tools on the market, its adoption could hardly be considered successful. As argued by Stevens (2010), the most probable reason for the failure is a set of major semantic issues. However, different users might have different reasons for abandoning it. A few will explicitly understand the problem with the semantics, and some will just see the difficulties of using it, since it is not well-supported by the tools –probably because of the semantic issues at first place. Moreover, building semantic foundations for QVT-R turned out to be a challenging issue. A formal semantics for a relatively simple *check-only* mode required rather intricate mathematical constructs based on symbolic graphs (Guerra and de Lara, 2012); formalization of the *enforce* mode is still an open issue. Thus, understanding of even a binary general synchronization is challenging, not to mention the multi-ary case. The lack of a sound underlying theory leads to a shaky conceptual framework, ambiguous terminology and flawed communication between tool users and tool builders, and ultimately to deficient tools.

We have studied the domain of synchronization scenarios of a network of models in more detail and the results of the study are presented in Chapter 2[1]. The study shows that there exist 44 different concrete synchronization types, and this number might even increase when we add further dimensions to the taxonomy. The variety of the synchronization types (as clearly revealed by the study) implies the possible variations in the definition and the maintenance of their corresponding model transformations. Such variation makes the task of defining a transformation and maintaining the model interrelations complex. To deal with this complexity, we need to employ suitable abstraction techniques that offer *declarative* methods with clear precise semantics in transformation definitions. The content of the current thesis is a contribution in this direction as will be explained further in the next section.

## 1.2 Aim of the Thesis

Examining the current MT languages in the literature reveals that they usually miss one or both of the important features of declarativity and formality, both of which are essential in dealing with the complexity of a network of models discussed in the previous section. While graph-based approaches (Ehrig *et al.*, 2006) have a sound formal foundation based on graph transformations, they unfortunately lack enough declarativity in terms of defining transformations at a higher level of abstraction (see Sect. 1.3). In contrast, more declarative approaches like ATL (Jouault and Kurtev, 2006), and QVT-R (OMG, 2015) lack enough formality in their foundations. Even aside from this restriction, as we will see in Chapter 5, the transformation definitions

[1]An extended version of this study is published by Diskin *et al.* (2016).

in these languages are still very involved with the element-wise manipulation of the model elements that makes them closer to imperative approaches.

The aim of the current thesis is to build a bridge between the engineering domain and the mathematical world in the context of MT approaches by further development of a Query-based Structured Model Transformation (QueST) approach that is formal and yet more declarative than the current MT approaches. The rich formality behind the QueST approach is already well developed in some previous works (Diskin, 1997, 1996, 2011; Diskin *et al.*, 2012; Diskin and Maibaum, 2012; Diskin and Dingel, 2006; Diskin, 2008). This thesis introduces diagrammatic query framework (DQF) that provides a foundation for the definition of collection-level diagrammatic query operations that are used in the definitions of QueST MTs. DQF is defined in Chapter 3 and explained in Chapter 4. Besides, it develops MT engineering by comparatively studying the QueST application to MT definitions in contrast to three rule-based MT approaches (i.e., ETL (Kolovos *et al.*, 2008), ATL (Jouault and Kurtev, 2006), and QVT-R (OMG, 2015)). Chapter 5 is focused on comparing QueST with these rule-based MT approaches showing QueST's relative advantages in terms of declarativity, modularity, and incremental design. Chapter 6 presents QueST's application in logical analysis of MT definitions by employing the formal language of Alloy (Jackson, 2002). Besides the QueST development, this thesis also contributes in the introduction of a 3D-taxonomical space of model synchronization scenarios mentioned earlier in Sect. 1.1.3. It motivates the necessity of declarative approaches in MT definitions by showing the variety of model synchronization scenarios. This work –though being presented in Chapter 2– can be studied before or after the study of the other chapters in this thesis.

In the following subsections, there is a brief elaboration on the aspects that will be presented in this thesis.

### 1.2.1 The QueST Approach

The basic idea of MT definitions in QueST is acquired from the successful approach of declaratively defining views in the relational databases. This way of defining transformations in MDE was initially proposed by Diskin and Dingel (2006) and later on developed more in Diskin's work (Diskin, 2008, 2011). Similar to the database view definitions, the MT definitions in QueST are specified over the metamodels (analogous to the schemata in the databases), and are executed over the models (analogous to the data in the databases). The view constructions in databases and metamodels are different as the corresponding underlying structures are represented differently (i.e., by relations in relational theory vs. by graphs in metamodeling). Furthermore, the definitions in QueST are accompanied by graphical representations that provide useful diagrammatic models assisting the users during MT developments. An elaborate discussion of the approach will be provided during this thesis, specifically in Chapter 4.

### 1.2.2 DQF Framework

Queries are the main ingredient of the QueST approach. They could be understood generally as a function retrieving some information from the source model to build the elements in the target model. Two main features of the QueST queries are as follows: 1) QueST queries are defined over the metamodel elements (collections) rather than model elements (elements). 2) They have diagrammatic arities (see Sect. 3.2.1, Definition 9) rather than ordinary tuple type arities. Queries are defined over metamodels; thus their semantic definitions are based on the semantics of the metamodels. Metamodel semantics can be defined either in a functorial setting or in a fibrational setting. In a functorial setting, the semantics of $M$ is defined as structure-preserving mappings from $M$ to some predefined universe (e.g., the category of sets and relations). In a fibrational setting, the semantics of $M$ is defined as structure-preserving mappings from a semantic domain (e.g., the category of graphs and homomorphisms) into M[1]. For example, semantics of a metamodel is given by (typing) mappings from data graphs into the metamodel graph. In Chapter 3, we will explain the metamodel semantics in a fibrational setting in more details. The formality behind QueST queries is previously discussed by Diskin (1997, 1996, 2011). In this thesis, we develop the QueST framework by introducing the DQF framework which enables the specification of the individual query definitions and executions more concretely. We aim to make the QueST approach more accessible to the MDE community. We will define the Diagrammatic Query Framework (DQF) in which queries are high-level collection-wise operations with diagrammatic arities. Queries are defined over a metamodel by matching their diagrammatic input arities over the metamodel graph, and augmenting the metamodel according to their corresponding output arities. We will explain the concepts around the framework using examples and will provide corresponding formal definitions in Chapter 4.

### 1.2.3 Comparative Analysis

Rule-based approaches, such as ETL, ATL, and QVT-R, are usually considered declarative approaches for defining model transformations. As this thesis promotes QueST as a declarative approach in MT engineering, Chapter 5 provides a comparative analysis between QueST and these rule-based approaches. QVT-R is chosen as it is

[1]This way of defining semantics is called *fibrational* semantic definition by Diskin and Wolter (2008)

the OMG standard approach to MT definitions. ATL is chosen as, to the best of our knowledge, it is the most well-known MT language widely used by the community. ETL is chosen as a consequence of our research visit to York University in the UK where it was developed alongside a series of EOL-based (Kolovos *et al.*, 2006) languages to support model management activities. We will show that the query operations in QueST are collection-level operations applied at the metamodel level instead of instance-wise manipulation of elements at the model level. In this sense, we say that QueST is more declarative than the studied rule-based approaches[1]. As an analogy, the query operations is QueST are close to the declarativity of the SQL (Date and Darwen, 1997) operations in contrast to record-by-record manipulation of data in the language of Network (Taylor and Frank, 1976) or Hierarchical (Tsichritzis and Lochovsky, 1976) databases. Besides this, our study shows that QueST offers more flexibility in the construction of an MT definition in terms of MT definition modularity and incremental development.

### 1.2.4 Logical Analysis of MT Definitions

Testing approaches, though having many advantages, usually can never guarantee the absolute conformance of an MT definition to the expected properties. Specifically, when an MT is involved in the development of critical systems, gaining additional confidence regarding the correctness of a transformation is strictly demanded.

With the exception of the graph-based languages (such as Triple-Graph-Grammars (Kindler and Wagner, 2007)), current MT approaches usually suffer from not providing enough formality that is essential for the logical analysis of MT definitions. For example, QVT-R encountered many problems in even precisely defining its underlying semantics (Stevens, 2010). Other languages like ATL do not initially come with formal definitions. This makes it very difficult (if not impossible) to logically analyze MT definitions that are defined in these approaches. Relying on the QueST formal semantics, we will show the way an MT definition in QueST can be encoded as a logical theory and the properties to be checked can be considered as propositions in the encoded specification language. Thus, using the mathematical tools, it will be possible to analyze the properties of QueST MT definitions. In this thesis, we used the formal language of Alloy (Jackson, 2012) to demonstrate the encoding and analysis techniques.

[1]In this thesis, we use declarativity to mean collection-level operations vs. element-wise operations. Employing this mechanism might, in turn, contribute to hiding explicit control structures.

## 1.3 Related work

In the following, we will first explore the approaches that are proposed for defining MTs in the MDE community. In the second part, we will explore the literature related to the proposed techniques for analyzing model transformations.

### 1.3.1 MT Approaches

We can generally place the proposed transformation languages in the MDE community into two major categories: 1) declarative rule-based approaches, 2) graph-based operational approaches. In the first category, the transformation is basically defined as a set of constructs called transformation rules that are usually executed nondeterministically over the source model; these rules have querying mechanisms that select elements from the source model based on which the elements in the target are constructed. ATL (Jouault and Kurtev, 2006), ETL (Kolovos *et al.*, 2008) QVT-R (OMG, 2015), and JTL (Cicchetti *et al.*, 2010) are some of the well-known examples from this category. All of these languages either use OCL (Warmer and Kleppe, 2000) or an OCL-like syntax to specify the queries. In the second category, a transformation is defined as a number of Graph Transformation (GT) rules (Ehrig *et al.*, 2006). These rules are executed on the source model until they are no longer applicable subject to certain defined execution policies. Triple Graph Grammars (TGG) (Schürr and Klar, 2008), DSLTrans (Barroca *et al.*, 2011), Story Diagrams (Fischer *et al.*, 2000), Henshin (Arendt *et al.*, 2010), MOLA (Kalnins *et al.*, 2005), and GReAT (Balasubramanian *et al.*, 2007) are examples of this category.

QueST proposes a different approach in the way a transformation is defined; it promises to be more declarative than rule-based approaches and uses query blocks rather than rules (see Chapter 5); however, at the same time it benefits from the formality of graph transformation approaches in its core semantic definitions (see Chapter 4).

As presented in Chapter 4, a QueST definition in the metamodel layer is built by defining queries and mappings that exist at the metamodel level. In the MT languages that employ the mapping idea, such as TGG, and QVT-R, the mappings are in fact defined at the instance level. In recent work, Freund and Braune (2016) demonstrate certain benefits of defining the mappings in the metamodel layer (similar to what QueST does) such as automatic conformance of the generated models to their corresponding metamodel which might be easily missed in rule-based and GT approaches. However, the work is more focused on providing a generic transformation algorithm that implements such metamodel-mappings.

In the following, we further explore the literature regarding the proposed rule-based and the graph-based model transformation languages.

### Rule-based MT Approaches

A detailed comparison between QueST and three of the well-known rule-based MT approaches will be provided in Chapter 5. However, since an important aspect of the rule-based approaches is their corresponding query language, we examine querying approaches further in this section.

The structure of a query language is heavily influenced by the underlying data model over which it is defined. For example, navigational features are necessarily supported by the query languages that operate on the graph data models. Relational databases require the corresponding query language to provide relational operations like relational Join and Union. From this perspective (i.e., the underlying data model of the query languages), we can categorize the query languages as the following: 1) Query languages for semi-structured data models 2) Query languages for structured data models. In the first category, the schema information is dynamic and accompanied with the main data (Buneman, 1997; Abiteboul *et al.*, 1997). This is the reason that a semi-structured data model is sometimes called "self-describing"[1]. For example, an XML document without a fixed schema corresponds to a semi-structured data model[2]. In the second category, the schema information is fixed and is usually well distinguished from the data itself. Relational databases (Codd, 1970) are the famous examples of these data models; the schemata of the databases that define the database structures are independently defined and separately maintained.

For each of the above categories, there exist many query languages. Lorel (Abiteboul *et al.*, 1997), UnQL (Buneman *et al.*, 2000), XQuery (Boag *et al.*, 2002), XML-QL (Deutsch *et al.*, 1999), and XIRQL (Fuhr, Norbert and Großjohann, Kai, 2001) are some examples of the first category. IncQuery (Bergmann *et al.*, 2010; Ujhelyi *et al.*, 2015), CYPHER (Kaur and Rani, 2013), SPARQL (Pérez *et al.*, 2006), OCL (Warmer and Kleppe, 2000), and SQL (Date and Darwen, 1997) are examples of the second category. Since metamodels –similar to relational databases- are structured, we examine the query languages of the second category a little bit more in the following. The queries in IncQuery specify graph patterns to collect model entities. The queries define the patterns similar to the methods used in the graph transformation approaches (Ehrig *et al.*, 2006). Cypher is a query language for the property graph data model (Robinson *et al.*, 2015); more specifically, it is used as the query language of the graph database Neo4j (Webber, 2012). Similar to IncQuery, Cypher also uses graph patterns to build the queries. SPARQL (Pérez *et al.*, 2006) is another language working similar to the above two query languages in terms of using graph matching

[1]Formally speaking, semi-structured data are modeled as some form of labeled, directed graphs.

[2]XML documents with predefined schemas and semi-structured data models are compared in Goldman *et al.* (1999).

patterns in constructing queries, instead. However, it is used to query RDF (Resource Description Framework) (Klyne and Carroll, 2004) documents. OCL (Object Constraint Language) (Warmer and Kleppe, 2000) is the OMG proposed standard for specifying constraints on models. However, its querying features such as navigating the models and building collections of objects, make it suitable to be used as a query language too. Finally, SQL (Date and Darwen, 1997) is a very well-known query language used for declaratively defining queries over relational databases.

From the above list of query languages, OCL or OCL-like languages are the ones that are currently used in the MT domain; for example, ETL, ATL, and QVT-R use OCL, and JTL uses an OCL-like language. Further, except for SQL, all the other query languages operate on the instance level. Thus, among all of the above query languages, SQL is seemingly the most qualified query language that has a sound formal basis and also is declarative enough to be used as an MT query language. But, there is, unfortunately, an important obstacle towards using SQL in MDE as its default underlying data model is relational schemas. QueST query operations that will be introduced in Chapter 4 are high-level declarative operations –similar to SQL operations– defined over metamodels rather than relational schemas.

### Graph-based MT Approaches

Although QueST does not fit in the graph-based MT category, we briefly explore the relevant literature and explain the techniques employed in this category with the hope of providing a better distinction between them and the QueST approach.

In all of the graph-based approaches that are mentioned earlier (see page 8), transformations are specified in terms of graph grammars. Among them, TGG (Kindler and Wagner, 2007) is the one that allows interpretation of these rule bidirectionally. In TGG, both the left-hand side and the right-hand side of graph rules are divided into three blocks: one of these blocks is associated with the source model, another is associated with the target model and the third one comprises nodes that relate elements from the latter two blocks. The forward and the backward transformation rules are derived automatically from these triple rules. Aside from TGG that has its idiosyncratic way of treating graph transformation rules, the main difference between the other approaches in this category is mostly concerned with the way they organize and apply the ordinary graph transformation rules in one direction. In DSLTrnas (Barroca *et al.*, 2011), the graph transformation rules are arranged in layers; the layers prioritize the rule applications; however, the rules within a layer are executed in a non-deterministic fashion. In MOLA (Kalnins *et al.*, 2005), the applications of the graph transformation rules are defined using programming constructs such as sequences, loops, and branching. GReAT (Balasubramanian *et al.*, 2007) supports parallelism and recursion besides the basic programming constructs. In Story Diagrams (Fischer *et al.*, 2000), the applications of the rules are controlled by structures similar to UML activity diagrams. Finally, Henshin (Arendt *et al.*, 2010) is used for transforming EMF (Steinberg *et al.*, 2008) models, and in addition to basic graph transformation features, it provides a variety of transformation units to define control structures for the rule applications.

### 1.3.2 MT Formal Analysis

Amrani *et al.* (2014) suggested identifying three aspects in approaching a model transformation verification problem. These aspects are transformation aspect (i.e., MT language and its features), kinds of properties to be checked, and the techniques to be used in an MT verification problem. Regarding the first aspect, we tend to consider QueST as a framework with a formal semantics rather than a concrete implementation language; nonetheless, QueST already provides a formal basis to be used for analysis purposes, as it will be shown in Chapter 6. The type of properties that we will study over QueST MTs are *Transformation Dependent* in the sense that the properties are regarding a specific transformation, and are *Input Independent* in the sense that they hold for any valid input model (see Amrani *et al.* (2014)). Finally, the technique that is used falls within the category of *theorem proving* techniques as we use a logical interpretation of MTs for analysis purposes.

Ab. Rahim and Whittle (2015) classified existing MT verification approaches. Regarding their coarse-grained classification, the approach presented in this thesis fits within the theorem proving (see Sect. 6.4.1) and model checking (see Sect. 6.1.4) approaches. With respect to their finer-grained classification, the QueST approach is considered formal; it automatically ensures type correctness[1] (see Sect. 4.3.2); it preserves the static semantics of models[2] (see discussion in Sect. 4.3.2 and as an example, Property 1 at Sect. 6.4.1), and the correspondence between source and target[3] can be formally verified (see, for example, Property 3 in Sect. 6.4.1).

Calegari *et al.* (2011) use a theorem proving technique in analyzing MT properties similar to our approach in Chapter 6. They presented an encoding of an ATL transformation as a logical theory in Coq (Bertot and Castéran, 2013): metamodels are translated to inductive types and model transformation rules to logical formulas, and properties specified as propositions in the corresponding theory; then they used the Coq theorem prover to analyze the properties. The differences mainly originate from the differences in the underlying transformation approach (i.e., ATL rules vs. QueST queries) and the encoding techniques; for example, they encoded metamodels

[1] "Elements in the target model are instances of elements in the target metamodel."
[2] "The target model conforms to well-formedness constraints of the target metamodel"
[3] "The target model contains elements that correspond to elements in the source model"

as inductive types in Coq while we encoded them as a family of sets and relations with constraints in Alloy. This affects the way axioms are expressed in the corresponding theories.

Büttner *et al.* (2012) provide an automatic translation of ATL transformations to transformation models expressed as OCL theories, and used an OCL model finder to analyze the properties. This work is similar to our work in this thesis with respect to using an instance finder (as we used the Alloy instance finder) to check MT properties.

Maude (Clavel *et al.*, 2002) is a high-level language that supports membership equational logic and rewriting logic. Romero *et al.* (2007) encoded models and metamodels in Maude. Troya and Vallecillo (2011) provided a formal semantics of ATL using Maude that enables accessing the Maude toolkit to reason about the encoded MT specifications. In contrast, we use Alloy for the MT logical encoding (see Sect. 6.1 for more details about Alloy); our MT logical encoding is also based on QueST rather than ATL.

Alloy is used by Macedo and Cunha (2013) to encode QVT-R transformations and provide alternative enforcement semantics for them. It is used by Cunha *et al.* (2013); Maoz *et al.* (2011) to translate the UML class diagrams to verify their corresponding properties. It is used by Anastasakis *et al.* (2007) to encode the source and the target metamodels and specify the transformation rules as mapping relations. Gammaitoni and Kelsen (2015) introduced a sub-language of Alloy called F-Alloy that is used for expressing a functional mapping representing a transformation. The latter two works (i.e., Anastasakis *et al.* (2007) and Gammaitoni and Kelsen (2015)) follow the common MT paradigm that a source-to-target transformation is specified as a set of mappings going from the source to the target elements, whereas in QueST these mappings go in the opposite direction.

## 1.4 Thesis Structure

Chapter 2 presents the taxonomy of model synchronization scenarios organized in a 3D space. The discussions in this chapter show the complexity of the model transformation domain, and the necessity of supplying MT with a more declarative and abstract approaches. This can be seen as a general motivation for the subsequent chapters. In Chapter 3, we provide the technical background for the framework, which is compactly defined in a formal style. Specifically, we define the notions of metamodel and model, and a diagrammatic query operation. Engineers interested in quickly seeing the QueST framework in action can skip the formal definitions and move to Chapter 4. In Chapter 4, we introduce the QueST approach in the definition of MTs using a simple example. We also explain the concepts behind the diagrammatic query framework (DQF) using examples. In Chapter 5, we compare queries in QueST and rules in the rule-based approaches –ETL, ATL, and QVT-R. We will use two examples –the HappyPeople example introduced in Sect. 4.1, and the standard ClassToTable example– to carry out the comparative analysis. In Chapter 6, we demonstrate the way a QueST MT definition can be logically verified against defined properties. We will define properties over the same two examples above, and analyze them using the Alloy analyzer and doing proofs manually. Finally, in Chapter 7, we conclude this thesis and mention future works.

Appendix A.1 presents an encoding of the QueST ClassToTable example in Alloy. Appendix A.2, Appendix A.3, and Appendix A.4 present the implementations of the ClassToTable example in, respectively, ETL, ATL, and QVT-R.

## 1.5 Thesis Contributions

The four contributions of the thesis are presented in four consecutive subsections below. Part (a) briefly describes the contribution, part (b) specifies the individual contribution of the author.

### A Taxonomy of Bidirectional Model Synchronization Scenarios

(a) This thesis provides a 3D taxonomical space of model synchronization scenarios. We investigate the characteristics of (binary) model synchronization scenarios from three perspectives: 1) informational symmetry, 2) organizational symmetry, and 3) incrementality. We argued that these dimensions are orthogonal, that is, their impact on the characteristics of a synchronization scenario is independent. Based on this, we built a 3D taxonomical space, in which 44 different types of synchronization are identified, and provided concrete examples instantiating each type and illustrating its properties. This work is presented in Chapter 2.

(b) An extended version of this work that includes the formalization of the synchronization scenarios is published by Diskin *et al.* (2016). The author of this thesis collaborated half and half with Diskin in this work except the excluded formalization part (that was mainly developed by Diskin[1]). This paper received contributions from the third and the fourth authors in terms of comments and reviews.

A preliminary version of this work was published by Diskin *et al.* (2014) as a conference paper. The author of the thesis contributed a quarter of that work. The journal version essentially extended the conference paper. It (i) studied in detail the process that we called linearization of synchronization dimension

[1]See the Appendix of the published work (Diskin *et al.*, 2016) for the formalization of the synchronization types.

(by which a simple partially ordered set is considered as linearly ordered), (ii) further investigated the mutual orthogonality of the dimensions, specifically the org×info and org×inc planes, (iii) refined the incrementality dimension to cover semi-incremental scenarios, (iv) explored a multitude of examples of synchronization scenarios. These all allowed us to refine the taxonomy to cover 44 rather than 16 scenarios identified by Diskin *et al.* (2014).

**Making the QueST Formalism Accessible to the MDE Community**

(a) The QueST formalism as developed by Diskin (1997, 1996, 2011) is a formal machinary. The current thesis contribution is to make this machinery accessible to the MDE community. To this end, the thesis contributed the following: 1) the QueST framework is completed by introducing the DQF framework which enables the specification of the metamodel augmentation mapping $\mathbb{S} \to Q(\mathbb{S})$ more concretely, 2) it specifies QueST in the languages accessible to the MDE community, and provides new examples of its application. The DQF framework is presented in Chapter 3, and the explanation of QueST is presented in Chapter 4.

(b) A Doctoral Symposium paper was published by Gholizadeh (2013), and several presentations were delivered within the NECSIS[1] project annual workshops (Gholizadeh, 2014, 2015). The DQF framework is not previously published.

**Comparative Analysis of QueST and Rule-based MT approaches**

(a) This thesis compares QueST with three commonly used rule-based MT approaches, namely ETL, ATL, and QVT-R. Based on this comparison, it presents the way MT engineering can benefit from the declarativity and incrementality features of the QueST approach. This work is presented in Chapter 5.

(b) A workshop paper by Gholizadeh *et al.* (2014) presents mainly the content of Sect. 5.3.4. The author of the thesis was the main contributor (approx. ninety percent) of this work. Other findings in this chapter are not published.

**Logical Analysis of MT via QueST and Alloy**

(a) MT approaches in practice usually lack enough formality in their foundations and this makes it difficult to mathematically analyze transformation definitions. This thesis shows the way QueST MT definitions can be encoded as a theory in the language of Alloy, and the way its properties can be logically analyzed. This work is presented in Chapter 6.

[1]Network on Engineering Complex Software Intensive Systems for Automotive Systems.

(b) A workshop paper published by Gholizadeh *et al.* (2015) explains the techniques using a simple example. The author of the thesis is the main contributor (approx. seventy-five percent) of that. Application of the method to the ClassToTable example is not published.

# Chapter 2

# Various Types of Synchronization Scenarios

Modern applications require a shift towards networks of models related in various ways, whose synchronization often needs to be incremental and bidirectional. This new situation demands new features from transformation tools and approaches, and a solid semantic foundation to understand and classify these features. A preliminary step addressing this problem is to study the domain of synchronization scenarios. In this chapter, we take three dimensions that affect the characteristics of synchronization scenarios: informational dimension, organizational dimension (i.e., directionality) and incrementality. We systematically study the characteristics of synchronization scenarios along these dimensions. Then, we examine the mutual interaction of these dimensions, and finally present a taxonomy of model synchronization scenarios, organized into a 3D-space. Each point in the space refers to a specific synchronization type. The taxonomy aims to help with identifying and communicating a proper specification for the synchronization problem at hand and for the available solutions offered by tools and MT approaches.

The technical discussions in this chapter provide further insights regarding the variety of synchronization scenarios and motivate the usefullness of the QueST approach as a formal declarative MT framework. However, the content of the remaining chapters can be studied independently of this chapter.

## 2.1 Introduction

Suppose that two models to be synchronized, $A$ and $B$, are given together with a consistency relation between them. Initially the models are consistent, and then one of them, say, $A$, is changed with an update $a\colon A \to A'$. The first question ($Q_1$) we need to ask is whether this update destroys consistency or not. In the latter case, we call the update *private*, and do nothing. In the former case, we call the update *public*, and need to act to restore consistency between models.

The action to maintain the consistency depends on the type of the update, and the *update propagation policy* we assume as given. If update $a$ is of a *propagatable type* (or just *propagatable*), we need to find an update $b\colon B \to B'$ on the other side, which changes model $B$ as minimally as possible but restores consistency (so that models $A'$ and $B'$ are consistent). We then say that update $a$ was propagated to the $B$-side, and call $b$ the result of propagation. The entire scenario is called *incremental* change propagation, or *incremental* synchronization. Importantly, incremental synchronization allows having a *public* update on one side, say, $a\colon A \to A'$ and, *concurrently*, a *private* update $b^*\colon B \to B^*$ on the other side. Then, after propagating $a$ to $b\colon B^* \to B'$, the consistency is restored and the updated private part of $B^*$ is preserved in $B'$.

However, it may happen that the interaction between models is organized in such a way that model $A$ is not allowed to make changes like $b$ on the $B$-side: if such a change is needed, it can only be achieved by an immediate editing of model $B$ rather than being propagated from the $A$-side. In this case, update $a$ must be rolled-back (or modified to a different update of a propagatable type). Thus, the second question ($Q_2$) we need to ask about update $a\colon A \to A'$ is whether it is propagatable or not according to the policy.

Finally, there are situations when consistency restoration is achieved by regenerating model $B'$ from scratch rather than by updating model $B$ (and no a priori given relationship between $B$ and $B'$ is assumed to hold). We call this scenario *non-incremental* change propagation, or *non-incremental* synchronization, from $A$ to $B$. Clearly, such a synchronization discards any private changes made on the $B$ side. Thus, the third basic question ($Q_3$) we need to ask about change $a\colon A \to A'$ is whether synchronization from $A$ to $B$ is incremental or not.

Questions $Q_i$, $i = 1, 2, 3$ and their duals, i.e., similar questions $Q'_i$ about synchronization from $B$ to $A$ originated with a change $b\colon B \to B'$, capture the most basic (we could also say *ontological*) features of the binary synchronization. Our idea of classifying binary synchronization is that different combinations of answers to these six questions determine *different* synchronization scenarios.

### 2.1.1 From Six Ontological Questions to a Taxonomy

Questions $Q_1$ and $Q_3$ (and their duals) have binary answers: either 'yes' or 'no' for each of them. Question $Q_2$ is more complex: in fact, we need to partition all possible public updates on the $A$-side into propagatable or not, and dually for the $B$-side to answer question $Q'_2$. Each such a partition can be classified with a *three-valued* scale depending on whether *all*, *some* or *none* of public updates on one side are propagatable to the other side. Thus, each of the questions $Q_2$ and $Q'_2$ has three possible answers for classifying a synchronization scenario. The total number of possible answer combinations (we will say *multi-answers*) is $2^2{\times}3^3{\times}2^2 = 144$, which means that we have a variety of 144 different types of synchronization scenarios. This would be an enormous taxonomy not easy to understand and use. In order to reduce the number of types, we will perform two analyses.

The first is to analyse each multi-answer w.r.t. its practical meaning: it may happen that a logically possible multi-answer is practically meaningless and hence the respective type can be excluded from the taxonomy. For example, choosing the answer 'none' for both $Q_2$ and $Q'_2$ would actually disallow any possibility of synchronization, and this multi-answer should be excluded. For another example, it does not make sense to consider incrementality (by answering question $Q_3$) for a non-existing update propagation (if $Q_2$ is answered 'none'). Later on, we will carefully analyze such situations and exclude practically meaningless multi-answers, which will significantly reduce the number of synchronization types to 76.

The second analysis aims to consider possibilities of merging several different multi-answers into one type, if the differences between the respective scenarios do not practically matter. We will see that, indeed, many pairs of multi-answers (and the respective types) are mutually convertible into each other by swapping $A$ and $B$ (we call such types *dual*). We will show that after replacing each pair of such types by one type (that we call *almost concrete*), the number of types is reduced to 44.

A flat taxonomy encompassing 44 types is still not easy to manage. To address the concern, we will develop a modularization mechanism by which 44 types will be grouped into 16 *abstract* types, each one containing from two to four (almost) concrete subtypes. Moreover, the set of these 16 types will be structured as a set of 16 points disposed at vertices, edges and face centers of a three-dimensional cube, so that each abstract type can be easily recognized by the triple of its coordinates (see Fig. 2.7 on page 34 where 16 abstract types are denoted by circles (with different frames); numbers inside circles show the number of (almost) concrete subtypes.

Each axis of the space embracing the cube presents a linearization of the multi-answer to a respective pair of ontological questions $(Q_i, Q'_i)$, $i = 1, 2, 3$ discussed above. Axis $X$ is a linearization of the mutli-answers to the pair $(Q_2, Q'_2)$, and we call the respective coordinate an *organizational symmetry*; it characterizes the organizational constraints to update propagations. Axis $Y$ is a linearization of mutli-answers to the pair $(Q_1, Q'_1)$, and we call the respective coordinate an *informational symmetry*; it characterizes relative information capacities of the models. (To be more accurate, of the model spaces in which the models evolve). Finally, axis $Z$ is a linearization of mutli-answers to the pair of questions about *incrementality* $(Q_3, Q'_3)$. In the next section, we explain how linearization works, and define the three axes. In Sect. 2.3 we join the axes together to build the cube, and discuss the modularization mechanism.

## 2.2 Three Dimensions of Model Synchronization

In this section, we discuss in detail the three dimensions of a binary synchronization: org-symmetry, info-symmetry and incrementality. For each of them, we will introduce a complete two-dimensional taxonomy classifying all logically possible cases. Then we linearize the multitude of cases, and place them on an axis with several coordinates. Later, in Sect. 2.3, we will build our 3D-taxonomic space from these axes.

### 2.2.1 Organizational Symmetry

In this section we consider *organizational* constraints to update propagation. A simple such constraint is when one of the models (say, $A$), is considered entirely dominating the other model $B$, so that consistency restoration always goes via propagating updates on the $A$ side to model $B$, and never in the opposite direction. This situation is common when a low-level model $B$ is generated from a high-level (abstract) model $A$.

**Example 1** (Compilation of Java programs)**.** *The dominated model B is bytecode generated from a Java program A. If the latter is changed, it is recompiled so that new bytecode is regenerated from scratch.* □

**Example 2** (Model compilation)**.** *The dominated model B is a Java program generated from a UML model A. When A is changed, new code is regenerated from scratch and overwrites the old one.* □

A more sophisticated version of this scenario is described next.

**Example 3** (Incremental code generation)**.** *Consider a UML tool that generates code stubs from class diagrams. When method signatures are changed in the class diagram, code must be regenerated in such a way that method signatures are updated to be in sync with the updated class diagram, but method bodies are to be preserved. That is, while code's public part (method signatures, class names, etc.) are updated to reflect changes in the UML model, code's private data—the method bodies—are kept untouched.*

*In more detail, suppose that we have a UML model* $A_0$*, and a Java program with stubs* $B_0$ *generated from* $A_0$*, which is later completed to state* $B'_0$ *by adding method bodies (see the upper half of the inset figure below, in which horizontal dashed lines indicate consistency, vertical arrows are updates, and the double line is identity).*

*Suppose that the UML models evolves to state* $A_1$*, and we need to regenerate code, but in such a way that method bodies (and other private data in* $B'_0$*) are preserved. Hence, rather than regenerating Java program* $B_1$ *from scratch, we generate an update*

*$B_0' \to B_1$ such that the private part of $B_0'$ is preserved, and $B_1$ is consistent with $A_1$. In other words, update $A_0 \to A_1$ on the A-side is propagated to update $B_0' \to B_1$ on the B-side. To complete the example, we assume that propagation of updates from the code to the model is prohibited.* □

In all examples above, model $B$ is either never changed manually and all its changes are propagated from the $A$ side, or, if $B$ can still be edited, the changes are not propagated to $A$, and are not allowed to destroy consistency with $A$ (otherwise they will be discarded by the next update from the $A$ side). We say that model $A$ *organizationally dominates* model $B$, and write $A >_{\mathsf{org}} B$. Of course, the dominance in Example 2 is different from the dominance in Example 3; later we will see that adding the *incrementality* dimension to the specification allows us to distinguish the two scenarios.

$$\begin{tikzcd}
A_0 \arrow[r, dash, dashed] \arrow[d, equal] & B_0 \arrow[d] \\
A_0 \arrow[r, dash, dashed] \arrow[d] & B_1 \arrow[d] \\
A_1 \arrow[r, dash, dashed] & B_2
\end{tikzcd}$$

Note that dominance is a property of an *ordered* pair of models, so that relations $>_{\mathsf{org}}$ and $<_{\mathsf{org}}$ are different (but mutually invertible); we will say that the case $A <_{\mathsf{org}} B$ (read "$A$ is *dominated by* $B$") is *dual* to $A >_{\mathsf{org}} B$ ("$A$ dominates $B$"). Both cases present a strictly asymmetric synchronization, and we refer to them as *org-asymmetric.* This corresponds to what is often called *one-way* or *unidirectional* transformation. We have an entirely different synchronization type for code generation with roundtripping.

**Example 4** (Roundtripping). *Let A be a UML model and B a Java code generated from A. We assume that changes are freely propagated in either direction based on the history: the* freshly updated *model, being either the UML model or the Java code, dominates and propagates its changes to the other side irrespectively to whether this freshly updated model is A or B.* □

In such a case, we say that neither model organizationally dominates the other, write $A >\!\!<_{\mathsf{org}} B$, and call this situation *organizational symmetry.* Note that relation $>\!\!<_{\mathsf{org}}$ is symmetric: it is equal to its inversion.

There are also important synchronization cases in-between strict asymmetry and strict symmetry considered above. A model can be *partially dominating* in the sense that *some* (but not *all*) updates on this model are allowed to propagate to the other side depending on the type of the update.

**Example 5** (Partial roundtripping). *Consider a modification of roundtripping Example 4, in which the synchronization policy only allows some code (model B) updates, e.g., changes in method signatures, to be propagated to the UML model A, whereas other code updates that violate consistency will be discarded.* □

We call the case above *organizational semi-symmetry* and write $A >\!\!\leq_{\mathsf{org}} B$. Note that the right part of the symbol indicates that only some updates on the $B$ side are propagated to $A$, while the left part of the symbol indicates that all updates on the $A$ side can be propagated to $B$. The entire relationship is phrased as *model A partially dominates B*, i.e., the direction of the entire relation is given by its "strong" half from $A$ to $B$.

The dual case is denoted by $A \geq\!< _{\mathsf{org}} B$ (read *model A is partially dominated by B*). It is realized in the following scenario with Java Development Tools (JDT) in Eclipse IDE.

**Example 6** (Outline view in JDT)**.** *The outline view A of some piece B of Java code is regenerated from scratch every time B changes. JDT also allows the user to make some selected operations in the outline view, e.g., renaming elements, or moving elements within the hierarchy. These updates are then propagated to the code. Thus, we have partial dominance* $A \geq\!< _{\mathsf{org}} B$. □

Note how inverting the relation of org-semi-symmetry from $A >\!\leq_{\mathsf{org}} B$ in *Example* 5 to $A \geq\!< _{\mathsf{org}} B$ in *Example* 6 dramatically changes the roles of models and their synchronization behavior.

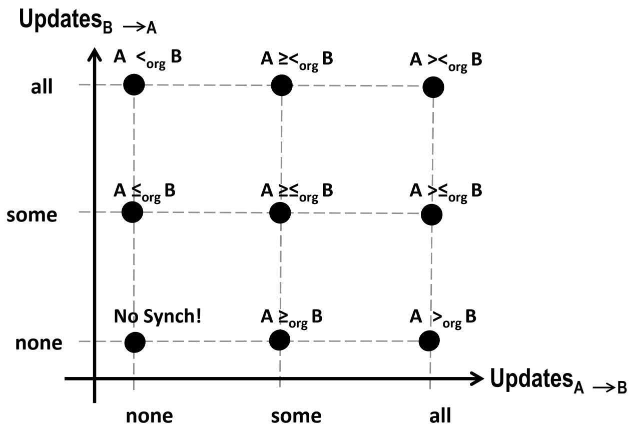
In fact, partial dominance (semi-symmetry) is common in model synchronization. A typical example is updatable (or editable) views in databases.

**Example 7** (Updatable views.)**.** *Let an application use an abstract view A of a data source B. All source updates can be propagated to the view, but the inverse propagation can be problematic (the infamous view update problem) because, in general, the same view state A can correspond to many different source states B: choosing any of them is not appropriate in the database context, in which data in B must properly reflect a real world state. However, some simple A- updates (e.g., those only affecting a direct projection of a source table) can be propagated to B in a unique way. Thus, the view is only partially dominated by the source,* $A \geq\!< _{\mathsf{org}} B$ *(while a non-updatable view is described by* $A <_{\mathsf{org}} B$*). Only for simple projection views, we may have* $A >\!< _{\mathsf{org}} B$. □

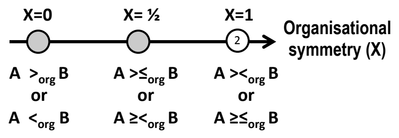
The examples we considered motivate a taxonomic plane shown in Fig. 2.1(a). The two axes of the plane correspond to the two directions of update propagation, and their coordinates indicate whether *none*, *some*, or *all* of the possible updates can be propagated in the direction that corresponds to the axis. Points in the plane are org-dominance types, which are instantiated by different synchronization cases, for example, those that we considered above.

So far, we have considered cases, whose type has at least one coordinate "all". Now we are going to discuss the other four types, and begin with the type *some-some* or $A \geq\!\leq_{\mathsf{org}} B$, for which both directions are sensitive to the update type, and only some updates can be propagated in either direction. We present two examples.

**Example 8** (Bi-partial roundtripping)**.** *Consider a symmetric modification of Example 5, in which the synchronization policy prohibits some model updates, e.g., changes in method signatures, to be propagated to the code, but these updates can be done on the code side and propagated back to the model.* □



a) Plane of org-dominance types


X=0
X= ½
X=1
Organisational symmetry (X)
A $>_{org}$ B or A $<_{org}$ B
A $>\leq_{org}$ B or A $\geq<_{org}$ B
A $><_{org}$ B or A $\geq\leq_{org}$ B

b) Its linearization

Figure 2.1: Organizational Dimension of Model Sync

**Example 9** (Class and sequence diagrams)**.** *Consider a system model consisting of a UML class diagram (CD) and a UML sequence diagram (SD) with the following synchronization policy. The structural part of the SD (a set of lifelines typed by classes) is generated from the CD, and class name changes are not allowed in the SD. Dually, the method signatures in the CD are generated from the SD, and changing them in the CD is not allowed.*

Thus, *some-some* org-symmetry, $A \gtreqless_{\mathsf{org}} B$, assumes a proper subset of propagatable updates on either side. Importantly, these two sets of updates are mutually complementary: any update on either side should be realizable by either a direct update on the side, or by propagating a suitable update from the other side (or by both like, e.g., Classes attribute names in *Example* 9). We will refer to this property as *completeness* of the $\gtreqless_{\mathsf{org}}$ symmetry. It holds for both of the examples above.

Now consider the mutually dual types, *none-some* or $A \leq_{\mathsf{org}} B$, and *some-none* or $A \geq_{\mathsf{org}} B$, when one side is an entirely passive receiver of updates from the other side, but only some updates can be propagated from the latter. An essential disadvantage of this synchronization is that some states are unreachable. For instance, in *Example* 9, if the *class diagram* was made an entirely passive, then class names cannot be updated at all as the *sequence diagram* is also not allowed to change them. We assume that such situation should be avoided, and we will exclude them from our taxonomy.

Finally, the type *none-none* means that models evolve independently without synchronization, and this case can be excluded from our taxonomy of *synchronization* types. Thus, there are four essentially different types of organizational dominance:

- (strict) *asymmetry* when one side is entirely dominated (or *passive*) and the other is entirely dominating (*active*);
- *semi-symmetry*, when the dominated side is not entirely passive and can propagate some updates;
- two *symmetric* cases, when both sides are either fully active, i.e., (*all-all*), or both are partially active, i.e.,(*some-some*); in the latter case we assume completeness so that any update can be reached.

Formally, the first two types appear as points $X = 0$ and $X = \frac{1}{2}$ on the *org-symmetry* axis in Fig. 2.1(b). Each of them is an *abstract* type in the sense that any case instantiating it actually instantiates one of the two *concrete subtypes* specified below the axis line. However, as each pair of such subtypes is mutually dual (the first subtype is converted into the second by swapping its arguments, and conversely), their distinction can be ignored and we say that types $X = 0$ and $X = \frac{1}{2}$ are *almost* concrete, or concrete *up to duality (of their subtypes)*. In contrast, the abstract type $X = 1$ has two essentially different concrete subtypes. We will refer to these two concrete subtypes as *rich* and *poor* org-symmetry (given by, resp., types *all-all* and *some-some* in the plane). If needed, we denote these subtypes by $X = 1^+$ and $X = 1^-$ resp. We also use shading to show whether the type is concrete or abstract: all types in the plane Fig. 2.1(a) are concrete and colored black; the symmetry type $X = 1$ in the axis Fig. 2.1(b) is abstract and blank; and asymmetric types $X = 0$ and $X = \frac{1}{2}$ are almost concrete and grey. The integers inside the blank circles (i.e., abstract types) specify the number of subtypes; we call them *multiplicities* (e.g., type $X = 1$ has multiplicity 2). We will use this notation and terminology throughout the paper.

### 2.2.2 Informational symmetry

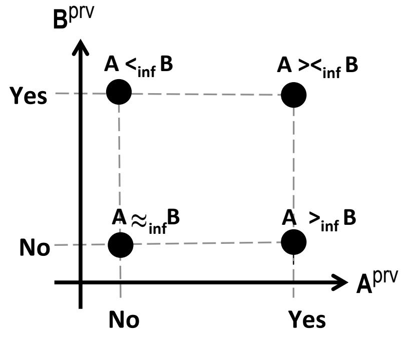
Informally, we say that a model $A$ is an abstract view of model $B$, if $A$ can be built from data provided by $B$. As in this process some information about model $B$ can be lost, different states of $B$ can result in the same state of $A$, i.e., different $B$'s can have the same view $A$. In this sense, we consider $A$ as being more abstract than $B$.

More formally, we first assume given some *consistency relation* $\mathsf{K} \subset \mathsf{M}\times\mathsf{N}$ over model spaces $\mathsf{M}$ and $\mathsf{N}$, in which models $A$ and, resp., $B$ reside (these spaces are determined by the respective metamodels, or grammars for textual models, e.g., code). If relation $\mathsf{K}$ is of one-to-many type, i.e., for any model $B \in \mathsf{N}$, there is exactly one $A \in \mathsf{M}$ such that $(A, B) \in \mathsf{K}$, but there may be many $B$ with $(A, B) \in \mathsf{K}$, then we say that $A$ is a *view* of *source* $B$, or that $A$ is *informationally dominated* by $B$, and write $A <_{\mathsf{inf}} B$.

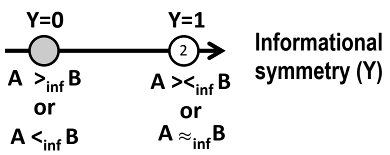
For instance, in *Example* 6 about JDT, the outline view $A$ is a view on source B in the formal sense above. Java program considered up to its executable content is a view of the byte code it generates. And in the code generation examples, Ex. 2 and 3, the UML model (considered up to its code generation context) is a view from which the source is generated: indeed, there may be multiple implementations of the same UML model.

It is convenient to think about informational dominance in terms of *private* and *public* (or shared) information. A view is entirely determined by the source and hence does not have a private part: any change of the view destroys its consistency with the source, and hence is a public update. In contrast, the source typically have private data (called "implementation details" for the case of code generation), which can be changed without affecting the view and thus are publicly invisible; in other words, the source has private updates.

The direction of model transformation may coincide with the direction of informational dominance as in the JDT example (i.e., Ex. 6); then we refer to the case as *view computation.* Alternatively, the direction of model transformation can be the opposite to info-dominance as in the (byte) code generation examples; then we say *source generation* or *view implementation.* The latter task is much more challenging than view computation because of an important constraint: the source is to be created in such a way that its public (shared) part ensures the given view.



a) Plane of info-dominance types



b) Its linearization

Figure 2.2: Informational Dimension of Model Sync

A more detailed analysis of synchronization scenario in Ex.2 shows that the UML model also has some private data. First, layout of boxes and arrows, time-stamps, etc. do not affect code generation and, thus, are private. Second, there may be structural differences between two UML models, e.g., in their inheritance hierarchies, which also result in the same code if the generator flattens the inheritance hierarchy. Then we consider inheritance as private data as well. Thus, each of the models (the UML model and the Java program it generates) has some private data not needed for the other model, and also some public data important for the other model (we may think of public data as shared but differently represented). We then write $A \asymp_{\mathsf{inf}} B$, and refer to the cases as *info-symmetry.*

In fact, info-asymmetry, say, $A <_{\mathsf{inf}} B$, is often a useful approximation of a symmetric situation $A \asymp_{\mathsf{inf}} B$, in which $A$'s private data are ignored. All our code generation examples are actually info-symmetric, but for many analyses it may be useful to view them as info-asymmetric by ignoring models' details not affecting the resulting code. We thus have two theoretical models of code generation: info-symmetric and info-asymmetric.

The plane in Fig. 2.2(a) classifies all possible cases of information relationships between two models. The two axes correspond to the two models, and their coordinates say whether the model does (Yes), or does not (No), have a private part (denoted by $A^{\mathrm{prv}}$, $B^{\mathrm{prv}}$ resp.). Types $A <_{\mathsf{inf}} B$ and $A >_{\mathsf{inf}} B$ are dual.

Type $A \approx_{\mathsf{inf}} B$ is a special case of info-symmetry, in which the consistency relation is one-to-one and determines a bijection between two model spaces as illustrated by the following example.

**Example 10** (HTML-MediaWiki)**.** *Consider synchronizing a wiki article described in a lightweight markup language like MediaWiki, and an equivalent HTML description of the article. Neither of two models has private data: both models are just different representations of the same information.* □

Thus, there are basically three different types of info-dominance:

- (strict) asymmetry when one side has private data whereas the other does not,
- two symmetric cases, when either both sides have private data (the *rich* info-symmetry), or neither one (the *poor* one)

Formally, we have two abstract types each partitioned into two concrete types as shown in Fig. 2.2(b) (we use the same notation as in Fig. 2.1b). If needed, we denote the concrete symmetric types $A \asymp_{\mathsf{inf}} B$ and $A \approx_{\mathsf{inf}} B$, by $1^{+}$ and $1^{-}$ resp.

### 2.2.3 Incrementality

The third dimension of our taxonomical space is *incrementality*, a well-recognized feature of model transformations. A *non-incremental* unidirectional model transformation $t\colon \mathsf{M} \to \mathsf{N}$ from a model space $\mathsf{M}$ to a space $\mathsf{N}$ creates a new target model

$B = t(A)$ from scratch every time model $A$ changes. An *incremental* model transformation is supposed to be more intelligent: a delta $a\colon A \to A'$ on the $A$ side is transformed into a respective delta $b = t_\Delta(a)\colon B \to B'$ on the $B$ side. Normally, the change caused by $b$ should be as minimal as possible, but it must restore consistency between $A'$ and $B'$.

In some synchronization scenarios, incrementality is optional and only used to improve efficiency. For example, incremental computation of the outline view in JDT in Ex. 6 may improve efficiency when dealing with very large code files. Similarly, efficiency of computing database views can be significantly improved, if they are computed incrementally. There are, however, situations in which incrementality is crucial; in these situations, a reasonable synchronization is essentially based on incrementality. For example, this is the case for the incremental code generation in Ex. 3, since losing method bodies with every change in the UML model that is propagated to the code is not desirable, incrementality is required. Such a situation is typical when updates are propagated to a side with private data, if the latter are to be preserved. For example, propagation of database view updates is necessarily incremental.

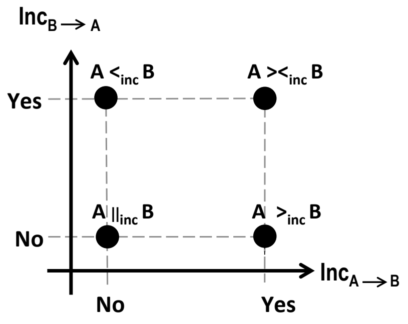
In more details, an incremental transformation $t$ takes a delta on one side, say, $a\colon A_0 \to A_1$, and the original model $B_0$ on the other side, and produces a delta on the other side, $t(a, B_0) = b\colon B_0 \to B_1$, which restores consistency between $A_1$ and $B_1$, and keeps the private part of $B_0$ unchanged in $B_1$. Deltas can be seen or even implemented as *traces* of what happened (or should happen) to individual model elements. If correspondences between models $A$ and $B$ are also precisely traced, an update propagation satisfying the requirements above can be assured (Diskin *et al.*, 2011b; Hermann *et al.*, 2011). In case that not all necessary traces can be provided (e.g., updates to code are often not tracked individually), a heuristics-based model-matching tool can be used to infer traces, which are then used for a proper synchronization. In this way, model comparison technologies can allow using synchronization tools that rely on traces with editing tools that do not provide traces.

In some synchronization scenarios, incrementality of change propagation in one direction is critical, while in the other direction is not.

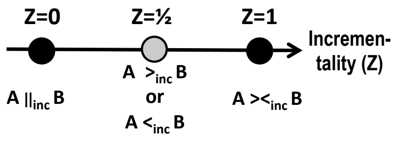
**Example 11** (Semi-incremental Roundtripping). *Method bodies' preservation, and hence incrementality of the respective change propagation, are critical, whereas preserving the layout of the class diagram in the reverse direction may be less important so that the class diagram is regenerated from scratch if code changes.* □

Of course, for large models comprising thousands of elements, synchronization as above is a logical possibility rather than a practically acceptable solution. However, the next example with database views is quite practical and, moreover, common.

**Example 12** (View maintenance). *View updates are always incrementally propagated: the source database cannot be arbitrarily "compiled" from the view (see also*



a) Plane of incrementality types



b) Its linearization

Figure 2.3: Incrementality of Model Sync

*Ex. 7). In contrast, view computation can be incremental to improve performance, or non-incremental, if performance is not an issue.* □

Thus, change propagation can be incremental in one direction and non-incremental in the other direction.

The plane in Fig. 2.3(a) classifies all possible cases. The two axes correspond to the two directions, and their coordinates say whether the changes are propagated incrementally (Yes) or not (No). Logically, this plane is similar to the information dominance plane Fig. 2.2: cases $A <_{\mathsf{inc}} B$ and $A >_{\mathsf{inc}} B$ are mutually dual, and there are basically three different types:

- (strict) asymmetry when one direction is incremental whereas the other is not; and
- two symmetric cases, when either both directions are incremental, or neither one is.

However, in contrast to linearization of info-dominance, we think that using our taxonomy would be easier if we keep the distinction between non-incrementality and bidirectional incrementality on the surface rather than hide it inside an abstract symmetric type. Hence, the incrementality plane is linearized as shown in Fig. 2.3(b): all types in the axis are (almost) concrete.

## 2.3 3D-Taxonomy of Model Synchronization

In this section, we discuss how the three dimensions work together, and join them into a 3D taxonomic space encompassing 44 types of synchronization behavior. We begin with a general discussion of different possibilities (perspectives) of viewing the space, and then consecutively consider three 2D-projections of the space, which are formed by three possible pairs of dimensions. We also develop a notational system providing a unique index for each of the 44 types, and illustrate its use with a table of examples for all types.

### 2.3.1 Different Ways of Viewing the Space

A 3D-space can be viewed in different perspectives: as the product of three separate axes, or as the product of a 2D-plane (say, $X \times Y$) and the respective orthogonal axis ($Z$). There are three such logical possibilities for our space, each one is based on a respective 2D-plane: Org×Info, Info×Inc, or Org×Inc (where we used short names for the axes). Each of these planes determines the corresponding perspective of viewing model synchronization, and each one is useful.

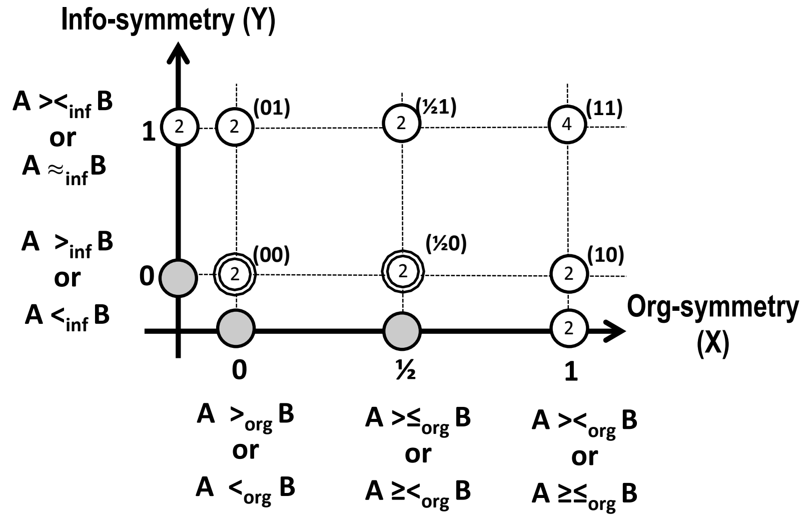
We may consider incrementality of change propagation as a secondary detail of a synchronization scenario, and classify a synchronization case by its position on the plane Org×Info. This will give us an incrementality-agnostic view of synchronization, which captures its most basic features, e.g., distinguishes view computation (abstraction) and source generation (refinement or implementation) as we will see in Sect. 2.3.2. Alternatively, we may consider incrementality as an important synchronization feature as incremental update propagation allows the receiving model to preserve its private data. In this sense, incrementality makes the receiving model less dependent on the dominant model, and hence classifying synchronization scenarios on the plane Org×Inc actually classifies dominance relations between the models, which is also a useful projection of a synchronization case. Finally, as demonstrated by the delta lens framework, the plane Info×Inc classifies the computational frameworks underlying model synchronization rather than synchronization scenarios as such. The latter are played out (according to their org-symmetry type) by using the change propagation operations of the underlying computational framework. Below, we will consecutively consider the three planes and the respective classifications; then we will discuss the whole 3D-space.

### 2.3.2 Org- $\times$ Info-Symmetry Plane

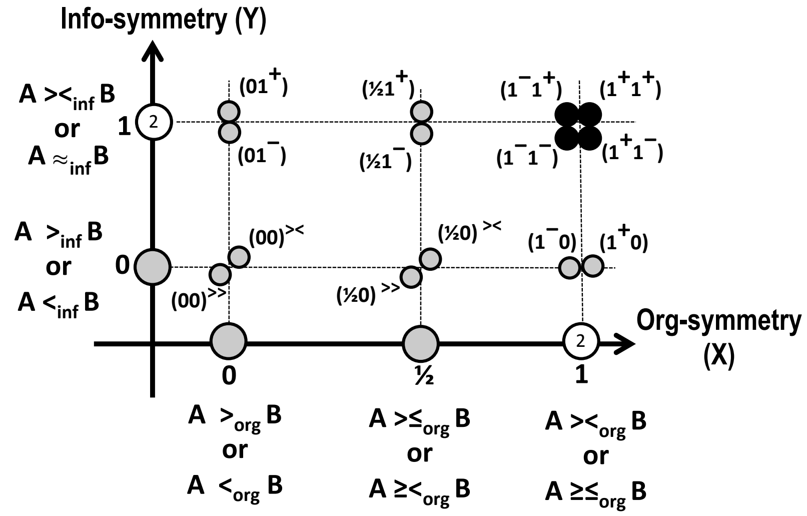
Recall the two cases of info-asymmetry, $A <_{\mathsf{inf}} B$, considered above. The first is when code $B$ is (either incrementally, *Example* 3, or non-incrementally, *Example* 2) generated from an UML model $A$ whose private data (i.e., those which do not affect code generation) are ignored. The second is when $A$ is an outline abstract view of code $B$ in Eclipse JDT (*Example* 6). Despite the same info-symmetry relationship between the models, synchronization behavior in the two cases is essentially different. In the former case, the view $A$ is active and generates the source that appears as a passive receiver of the view updates, $A >_{\mathsf{org}} B$. In the latter case, the view is mostly a passive receiver of the source updates, $A <_{\mathsf{org}} B$ (or $A \geq\!<_{\mathsf{org}} B$, if some updates can be propagated from the view to the source). What determines the synchronization type of the case is the combination of two parameters: the org- and the info-symmetry of the models. These two parameters are independent, and can be considered as two orthogonal coordinates forming the plane in Fig. 2.4(a).

The axes $X$ and $Y$ are, resp., the org- and info-symmetry dimensions specified in Fig. 2.1(b) and Fig. 2.2(b). Correspondingly, the plane has six taxonomic points or *types*. They are all abstract, and their multiplicities (integers inside the circles) show how many (up to duality) concrete subtypes they have. For example, type (11) in Fig. 2.4(a) has four concrete subtypes formed by all possible combinations of the concrete subtypes of its coordinates (note the cluster of four black bullets (concrete types) in the right top corner of Fig. 2.4(b)). Each of these concrete subtypes can be denoted by expression $(1^i1^j)$ with indexes $i, j$ ranging over set $\{+, -\}$ in which '+' denotes the richer symmetry (*(all, all)* and *(yes, yes)* in, resp., the org-plane in Fig. 2.1 and the info-plane in Fig. 2.2, and '$-$' refers to the poorer symmetry (*(some, some)* and *(no, no)* respectively). For example, code generation with full roundtripping *Example* 4 is a org- and info-symmetrically rich scenario of type $(1^+1^+)$ (we assume that private data of the UML model is not ignored by the synchronization). However, synchronization of UML class and sequence diagram *Example* 9 has the poor org-symmetry, and hence its type is $(1^-1^+)$.

Similarly, abstract types (01), $(\frac{1}{2}1)$, and (10) in Fig. 2.4(a) also have four concrete subtypes, but these are formed by two pairs of mutually dual subtypes, which we do not usually need to distinguish (see the discussion of duality in Sect. 2.2.1). Hence, we consider these types as having two (almost) concrete subtypes, and their multiplicities are set to 2. These subtypes are denoted by expressions $(1^i0)$ or $(x1^i)$ with $x \in \{0, \frac{1}{2}\}$ and $i \in \{+, -\}$ (see Fig. 2.4(b)) with the same meaning of $+$ for rich, and $-$ for poor, symmetries resp. For example, partial roundtripping with model's private data taken into account is typed by $(\frac{1}{2}1^+)$. For an instance of type $(\frac{1}{2}1^-)$, consider the wiki example Ex. 10, when in HTML only text changes are allowed whereas layout changes can only be made in MediaWiki.



a) Abstract Types



b) (Almost) Concrete Types

Figure 2.4: Plane of org- $\times$ info-symmetry

Abstract types (00) and ($\frac{1}{2}0$) are also split into two concrete subtypes, but owing to a different mechanism—*interaction of asymmetries*. Consider, for example, type (00), and note that synchronization behaviors of cases ($A <_{\mathsf{org}} B, A <_{\mathsf{inf}} B$) and ($A <_{\mathsf{org}} B, A >_{\mathsf{inf}} B$) are essentially different: the former is a (relatively simple) view computation, the latter is a (typically complex) source generation. The same mechanism of asymmetries interaction works for type ($\frac{1}{2}0$). In our taxonomic diagrams, we will distinguish abstract types split into different concrete subtypes due to asymmetries interaction by double-framing (to recall two asymmetric relations behind the type).

For denoting subtypes created by asymmetries interaction, it is convenient to use the following terminology and notation. If the same model dominates in both relations constituting a subtype of abstract type ($xy$), we say that relations are *equally directed* and denote the subtype by $(xy)^{\gg}$. If the dominant models are different, we say that relations are directed *oppositely* and denote the subtype by $(xy)^{\rtimes}$ (see Fig. 2.4(b)). For example, view computation is typed by $(00)^{\gg}$ if the view is not updatable, and by $(\frac{1}{2}0)^{\gg}$ if some view updates are possible. In contrast, model compilation is typed by $(00)^{\rtimes}$, and code generation with partial roundtripping is typed by $(\frac{1}{2}0)^{\rtimes}$.

We call the taxonomic planes in Fig. 2.4(a) and (b) *abstract* and *concrete* resp. We will also say that the former is an abstract *interface* to the latter.

### 2.3.3 Org-Symmetry $\times$ Incrementality Plane

Pairing org-symmetry and incrementality produces the taxonomic plane in Fig. 2.5. We interpret indexes $Z = 0, \frac{1}{2}$ and 1 as none, one, and, resp., all of the *propagatable* directions are incremental. Hence, when we have an org-asymmetric case $X = 0$ so that only one direction is propagatable, the case is classified by (00) if this direction is non-incremental, and by (01) otherwise. Type ($0\frac{1}{2}$) is excluded to avoid double typing. Other 2D types are formed by straightforward "multiplication" of the respective 1D types (points on the axes), and can be denoted in the same way as we did above (e.g., the central double-framed abstract type has two concrete subtypes formed by two possible combinations of directed relations: the same model dominates in both dimensions or each model dominates in exactly one dimension).

The plane in Fig. 2.5 is convenient for classifying practical scenarios, in which the info-symmetry is not a primary concern. For example, all code generation scenarios mentioned above can be treated in either the info-symmetric way, or info-asymmetrically, if we ignore private data (layout, timestamps, etc.) of the UML model.

Several examples are in order. Model compilation (non-incremental by default) in *Example* 2 is of type (00), whereas incremental code generation in *Example* 3 is

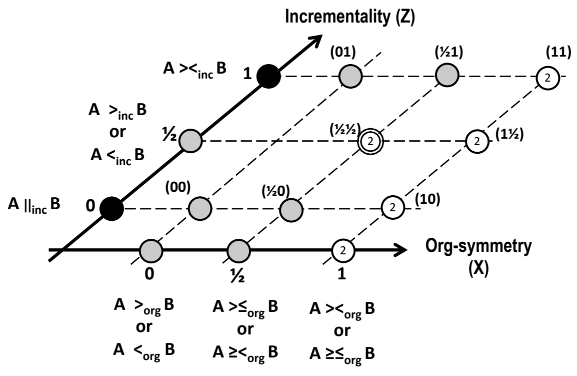

Incrementality (Z)
(01)
(½1)
(11)
A ><$_{inc}$ B
1
A >$_{inc}$ B or A <$_{inc}$ B
½
(½½)
(1½)
A ||$_{inc}$ B
0
(00)
(½0)
(10)
Org-symmetry (X)
0
½
1
2
A >$_{org}$ B or A <$_{org}$ B
A >≤$_{org}$ B or A ≥<$_{org}$ B
A ><$_{org}$ B or A ≥≤$_{org}$ B

Figure 2.5: Org-symmetry $\times$ Incrementality Plane

of type (01). Roundtripping in *Example* 4 can be (a) incremental in both directions, which gives us type $(1^{+}1)$; (b) semi-incremental, as in *Example* 11, which is of type $(1^{+}\frac{1}{2})$; or (hypothetically) (c) fully non-incremental when both models and code are "compiled" into each other, which is of type (10). Incremental code generation with partial roundtripping (*Example* 5) will be of type $(\frac{1}{2}1)$, if code changes are also propagated incrementally, and of type $(\frac{1}{2}\frac{1}{2})^{\gg}$ otherwise. For database view maintenance (*Example* 12), source computation is always incremental. If the view is non-incrementally computable and partially updatable, its type is $(\frac{1}{2}\frac{1}{2})^{\rtimes\!\!\ltimes}$. Making the view computation incremental changes the type to $(\frac{1}{2}1)$, and making it, in addition, non-updatable results in type (01).

### 2.3.4 Info-symmetry $\times$ Incrementality Plane

Recall that code generation can be realized non-incrementally (*Example* 2) and incrementally (*Example* 3), and each of these realizations can be considered either info-symmetrically or info-asymmetrically (when private details of UML models are ignored). Hence, the absence or presence of incrementality can be seen as a new dimension orthogonal to info-symmetry, and together they form an abstract taxonomic plane in Fig. 2.6.

Axis Y is the info-symmetry axis introduced in Fig. 2.2(b), and axis Z is the incrementality axis from Fig. 2.3(b). Their product provides six types shown in the figure, where we use notation introduced above for Fig. 2.4. Two types are almost concrete (grey), while the rest four are abstract with multiplicity 2. Note the double-framed type, whose two distinct (almost) concrete subtypes, $(0\frac{1}{2})^{\gg}$ and $(0\frac{1}{2})^{\rtimes\!\!\ltimes}$, are formed by the interaction of asymmetries similarly to the ones discussed before for Fig. 2.4.

Subtype $(0\frac{1}{2})^{\gg}$ encodes the case when both relations are equally directed (i.e., one side is info-dominant and its change propagation is incremental), whereas type $(0\frac{1}{2})^{\rtimes\!\!\ltimes}$ encodes the case they are directed oppositely. For example, incrementally

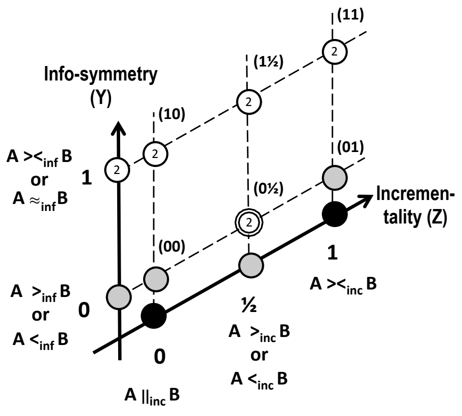


Figure 2.6: Info-symmetry × Incrementality Plane

computed non-updatable view is of type $(0\frac{1}{2})^{\gg}$, while non-incremental but updatable view is of type $(0\frac{1}{2})^{><}$. We delay considering more examples until the 3D-space will be introduced in Sect. 2.3.5, because examples without their organizational coordinate are not very interesting.

## 2.3.5 3D-space and Its Indexing

Although important and useful, 2D-projections of the space lack important aspects of synchronization if considered separately. For example, the Org×Info plane is fundamental and allows to distinguish view computation from source generation, but as we have seen, incrementality constitute an important complement to the organizational dimension and also affects dominance relations between the models. For example, classifying dominance without incrementality does not allow us to distinguish between model compilation and incremental code generation. On the other hand, classifying dominance relations by their type in the Org×Inc plane is also incomplete as incrementality affect dominance only if the dominated model has private data, whose presence or absence is specified by the third direction (info-symmetry). And, as we stressed several times, the plane Info×Inc only classifies the computational frameworks rather than synchronization scenarios as such—classifying the latter needs the organizational dimension. Thus, we do need to consider the entire 3D-space Org×Info×Inc for classifying scenarios within our 3D-framework.

As we will see, the 3D-taxonomy comprises 44 synchronization types, whose multitude requires a modular presentation, and a convenient notation. Particularly, we need a notation that allows unique indexing of the types. As a modularization mechanism to make the taxonomy manageable, we use the idea of having abstract types comprising several concrete subtypes as we did it when considering 2D-planes. An abstract interface to the space is shown in Fig. 2.7 (we will refer to it as the *cube*), where circles denote abstract synchronization types. Multiplicities inside circles show

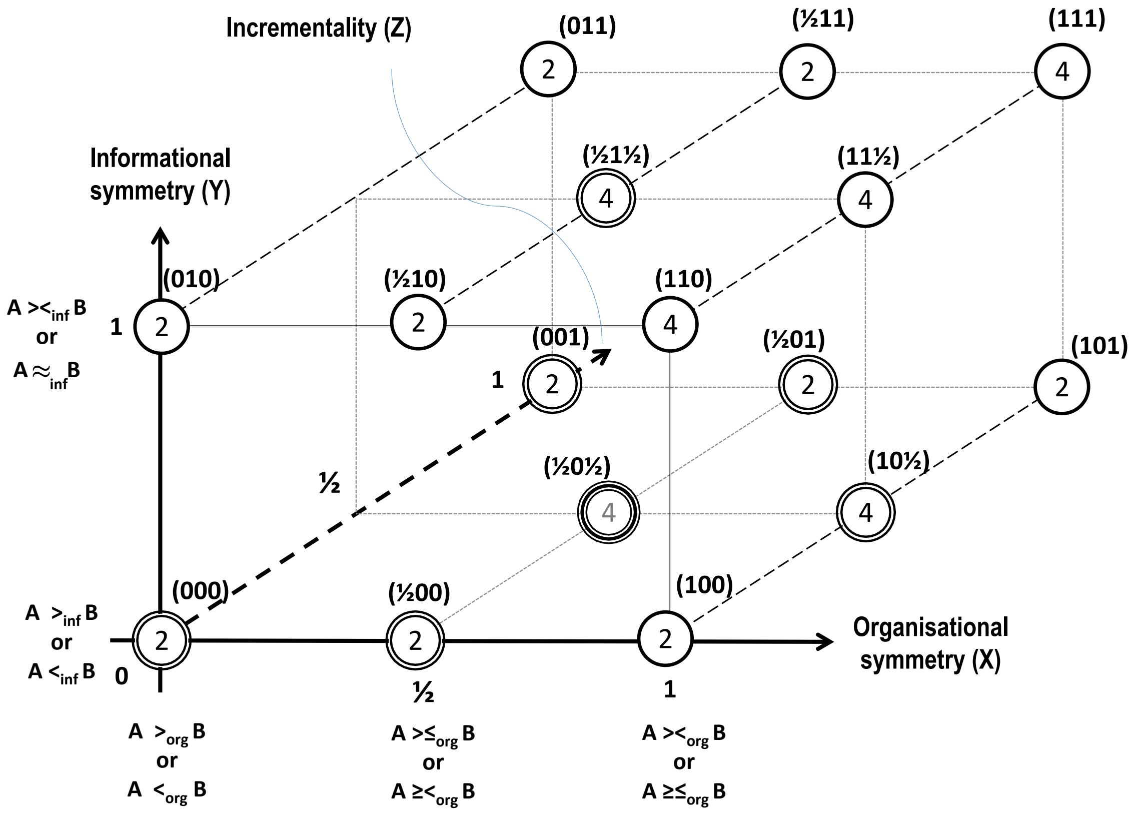

Incrementality (Z)
Informational symmetry (Y)
Organisational symmetry (X)
(000) 2
(½00) 2
(100) 2
(010) 2
(½10) 2
(110) 4
(001) 2
(½01) 2
(101) 2
(011) 2
(½11) 2
(111) 4
(½0½) 4
(10½) 4
(½1½) 4
(11½) 4
0
½
1
½
1
1
A $>_{inf}$ B or A $<_{inf}$ B
A $><_{inf}$ B or A $\approx_{inf}$ B
A $>_{org}$ B or A $<_{org}$ B
A $>\leq_{org}$ B or A $\geq<_{org}$ B
A $><_{org}$ B or A $\geq\leq_{org}$ B

Figure 2.7: Taxonomic space of synchronization types

the number of concrete subtypes up to their mutual duality. Below we will first discuss how two subtyping mechanisms introduced above for 2D-planes, and a notational system for subtype indexing, work for the 3D-space; then we will consider several examples of 3D-classification. The section culminates with a table illustrating each type with an example of synchronization.

### Subtyping Mechanisms 1: Rich vs. Poor Symmetries

One subtyping mechanism (we also say *type splitting*) is due to different possible combinations of *poor* and *rich* symmetries. The corresponding abstract types are single-framed: these are all types in the upper face of the cube except the central one, and two lower symmetric types (100) and (101) in Fig. 2.7 (recall that both incrementality values $z = 0$ and $z = 1$ denote symmetric cases without subtypes). Each of the subtypes can be denoted by superindexing the corresponding symmetric index 1 with plus(+) for the rich symmetry, or minus(−) for the poor one, as it was explained for Fig. 2.4(b). For example, if we consider code generation and do not ignore the UML model's private data, then model compilation is of type $(01^{+}0)$; fully incremental roundtripping is of type $(1^{+}1^{+}1)$; and semi-incremental roundtripping is $(1^{+}1^{+}\frac{1}{2})$. If we ignore the model's private data, all the types above remain the same except that the second coordinate changes to 0.

### Subtyping Mechanisms 2: Interaction of Asymmetries

The second cause of abstract type splitting is the interaction of two asymmetric (i.e., directed) relations. 3D-types formed by multiplying two directed relations are denoted by double frames: there are five such types in the lower face of the cube (i.e., $(000), (\frac{1}{2}00), (10\frac{1}{2}), (001), (\frac{1}{2}01)$), and one is in the upper face (i.e., $(\frac{1}{2}1\frac{1}{2})$). The type $(\frac{1}{2},0,\frac{1}{2})$ in the lower face is the product of three directed relations, and hence triple-framed; we will consider it in the next subsection.

For all double-framed abstract types $x0z$ in the lower face, either the org-symmetry value $x$, or incrementality value $z$, denote a symmetric case, and hence interaction of type $xz$ with asymmetric relation $y$ is simple. When a 3D-type $xz \times y$ is formed, and the direction of $y$ coincides with the direction of the asymmetric component in $xz$, we have a subtype denoted by $(xyz)^{\gg}$. If the directions are opposite, the subtype is denoted by $(xyz)^{\rtimes}$. For example, partial roundtripping, when both code generation and backward model updates are incremental, is of type $(\frac{1}{2}01)^{\rtimes}$.

The multiplicity of the type $(10\frac{1}{2})$ is 4 because two types with different asymmetry interaction are multiplied by two concrete types of org-symmetry, $A \rtimes_{\mathsf{org}} B$ and $A \underline{\rtimes}_{\mathsf{org}} B$. Splitting of the double-framed type in the upper face is analogous to splitting of the central type $(\frac{1}{2}\frac{1}{2})$ in Fig. 2.5, and subtypes are denoted in a similar way with the info-symmetry index added, i.e., by $(\frac{1}{2}1^{i}\frac{1}{2})^{\rtimes}$ if the directions are opposite, and $(\frac{1}{2}1^i\frac{1}{2})^{\gg}$ otherwise (with $i \in \{+,-\}$ depending on whether the info-symmetry is *poor* or *rich*).

It is useful to remember that *rich* or *poor* symmetries are indexed by superscripting the respective 1-coordinate within the 3D-type at hand, whereas the type of asymmetry interaction is indexed by superscripting the entire 3D-type.

**Subtyping Mechanisms 3: Triple-asymmetry Interaction**

The triple-framed type $(\frac{1}{2}0\frac{1}{2})$ is formed by multiplying the type $(\frac{1}{2}\frac{1}{2})$ in the plane $XZ$ (Fig. 2.5) by the (*almost*) concrete info-asymmetric type $Y = 0$. Due to the interaction of asymmetries, this multiplication consists of four subtypes. If the directions of org- and info-asymmetry coincide, and incrementalty goes in the opposite direction, we have the type $(\frac{1}{2}0\frac{1}{2})^{\gg<}$, while if all three directions coincide, the type is $(\frac{1}{2}0\frac{1}{2})^{\gg>}$. Note that the order of the inequality symbols in the triple superindex coincides with the order of dimensions (org-sym, info-sym, increm).

If the directions of org- and info-asymmetry are opposite, and hence give us an abstract type $(\frac{1}{2}0\frac{1}{2})^{><}$, we again have two concrete subtypes: (i) $(\frac{1}{2}0\frac{1}{2})^{><>}$, if the direction of incrementality coincides with the org-asymmetry direction, and (ii) $(\frac{1}{2}0\frac{1}{2})^{><<}$, if the incrementality direction coincides with info-asymmetry.

**3D-Classification at Work: Examples**

With the notation introduced above, all 44 concrete (up to duality) types are provided with a unique and, hopefully, suggestive index. Its general format is $(x_1^{s_1}x_2^{s_2}x_3)^{\alpha}$, where $x_i$, $i = 1,2,3$ is a dimension, and its superindex $s_i \in \{+,-\}$ appears near a coordinate $x_i$, $i = 1,2$ iff the coordinate is 1 and hence denotes the respective symmetry; then $s_i$ takes the value $+$ for the rich symmetry, and value $-$ for the poor symmetry. The superindex $\alpha$ denotes the type of asymmetry interaction and appears iff at least two of the three coordinates take their values from set $\{0, \frac{1}{2}\}$, i.e., at least two of the three inter-model relations are asymmetric (and hence directed). If exactly two coordinates are such, then $\alpha$ is $\gg$ if the two directions coincide, and $\alpha$ is $><$ otherwise. If all three coordinates are directed, then $\alpha$ takes its value from set $\{\gg>, \gg<, ><>, ><<\}$ according to rules in Sect. 2.3.5.

To illustrate how our 3D-indexing works, we will begin with considering several examples classified by four concrete subtypes of the abstract type $(\frac{1}{2}0\frac{1}{2})$ at the center of the bottom face in Fig. 2.7. We will also discuss some of the logically possible but not very practical synchronization types.

A simple example of $(\frac{1}{2}0\frac{1}{2})^{\gg<}$ type is a non-incrementally maintained and partially updatable database view (Ex. 12, recall that view updates are propagated incrementally by default). It is also easy to see that view maintenance of type $(\frac{1}{2}0\frac{1}{2})^{\gg>}$ means that the direction of the incrementality changes: view maintenance is incremental, but view updates (if allowed) will regenerate the source from scratch. Of course, although logically possible, such synchronization policy would be prohibited in the database world. An MDE counterpart of this example is an outline JDT view of code with a non-incremental option of code generation (Ex. 6), which seems to be a bit less prohibitive, but also practically non-desirable scenario that allows overwriting code changes. Thus, type $(\frac{1}{2}0\frac{1}{2})^{\gg>}$ classifies practically non-desirable scenarios.

For exemplifying types $T_1 = (\frac{1}{2}0\frac{1}{2})^{><>}$ and $T_2 = (\frac{1}{2}0\frac{1}{2})^{><<}$, we can take the following two versions of the partial roundtripping example Ex.5: version $(E5_1)$, in which only code generation is incremental, and version $(E5_2)$, in which only code-to-model update propagation is incremental. The latter case is again practically less desirable: while the public part of the code update will be propagated to the model and hence preserved in the system, the private part of the code update will be overwritten with the next update propagated from the model. In contrast, example $(E5_1)$ is practically reasonable as the info-symmetry index $Y = 0$ shows that we ignore the model's private data.

There are other logically possible but not practically desirable types. For example, such is type $T_3 = (\frac{1}{2}1^{+}\frac{1}{2})^{\gg}$, for which the info-dimension is symmetric, and hence the superindex $\gg$ specifies the interaction of org-asymmetry and uni-directional incrementality, and shows that incrementally propagated updates go from the org-dominating to the org-dominated side. Indeed, we note that the type allows both sides to be updated, but one direction of change propagation is non-incremental, and hence private data on the target side will be overwritten with every synchronization. For example, for partial roundtripping (Ex.5) of type $T_3$, the code-to-model direction is not incremental, and hence private data of the model will be lost when code changes are propagated to the model.

Table 2.1 provides illustrating examples for all 44 (almost) concrete types in the taxonomy; the rightmost column enumerates all the examples for referencing. The rows in the table are grouped w.r.t. their values for the three dimensions: for example, the organizational dimension for the first eight rows is 0 (see column X), which means that all examples in these rows are unidirectional. The rows within such a group are further sub-grouped w.r.t. their info-symmetry (column Y), and the third sub-grouping is based on incrementality (column Z). Finally, the Subtype column gives the unique index of an individual type, whose instance is given by the example in the respective row (see Sect. 2.3.5-2.3.5 for our indexing notation). Rows 17, 19, 21, 23 present pairs of mutually dual cases.

The examples are mainly based on the code generation scenario in MDE, except several cases of poor info-symmetry, illustrated with HTML-MediaWiki (Ex. 10) variations. Rows for the types that classify practically "dangerous" scenarios, in which private data can be lost as explained above for type $T_3$, are shaded. We also remind that incrementality enters the asymmetry interactions only if its index $Z = \frac{1}{2}$.

Table 2.1: Examples for each synchronization subtype in the 3D space. For code generation examples, “*-interpretation” means that model’s private data are **not** ignored for classifying the example.

| X | Y | Z | Subtype | Example | |
|---|---|---|---|---|---|
| Org = 0 | Inf = 0 | 0 | $(000)^{><}$ | Model compilation (Ex. 2), Compilation of Java programs (Ex. 1). | 1 |
| | | | $(000)^{>>}$ | Outline view in JDT (Ex.6): the view is non-editable generated non-incrementally. | 2 |
| | | 1 | $(001)^{><}$ | Incremental code generation (Ex. 3). | 3 |
| | | | $(001)^{>>}$ | Outline view in JDT (Ex.6): the view is non-editable generated incrementally. | 4 |
| | Inf = 1 | 0 | $(01^{+}0)$ | Model compilation (Ex. 2) in the *-interpretation. | 5 |
| | | | $(01^{-}0)$ | HTML-MediaWiki (Ex.10): MediaWiki generation from HTML is non-incremental and unidirectional. | 6 |
| | | 1 | $(01^{+}1)$ | Incremental code generation (Ex. 3) --the *-interpretation. | 7 |
| | | | $(01^{-}1)$ | HTML-MediaWiki (Ex.10): MediaWiki generation from HTML is incremental and unidirectional. | 8 |
| Org = ½ | Inf = 0 | 0 | $(\frac{1}{2}00)^{><}$ | Partial roundtripping (Ex. 5): both directions are non-incremental. | 9 |
| | | | $(\frac{1}{2}00)^{>>}$ | View maintenance (Ex. 12): partially updatable views that both directions are non-incremental. | 10 |
| | | ½ | $(\frac{1}{2}0\frac{1}{2})^{><>}$ | Partial roundtripping (Ex. 5): only code generation is incremental. | 11 |
| | | | $(\frac{1}{2}0\frac{1}{2})^{><<}$ | Partial roundtripping (Ex. 5): only code to model is incremental. | 12 |
| | | | $(\frac{1}{2}0\frac{1}{2})^{>><}$ | View maintenance (Ex. 12): non-incrementally maintained and partially updatable views. | 13 |
| | | | $(\frac{1}{2}0\frac{1}{2})^{>>>}$ | View maintenance (Ex. 12): non-incremental source generation and partially updatable views (source gen inc.)<br>Outline view in JDT (Ex. 6): the view is partially editable, code update is non-incremental but view update is incremental. | 14 |
| | | 1 | $(\frac{1}{2}01)^{><}$ | Partial roundtripping (Ex. 5): both directions are incremental. | 15 |
| | | | $(\frac{1}{2}01)^{>>}$ | View maintenance (Ex. 12): partially updatable views that both directions are incremental. | 16 |
| | Inf = 1 | 0 | $(\frac{1}{2}1^{+}0)$ | Partial roundtripping (Ex.5) in the *-interpretation: both directions are non-incremental. | 17 |
| | | | $(\frac{1}{2}1^{-}0)$ | HTML-MediaWiki (Ex.10): text updates on the HTML side are prohibited, both directions are non-incremental. | 18 |
| | | ½ | $(\frac{1}{2}1^{+}\frac{1}{2})^{><}$ | Partial roundtripping (Ex.5) in the *-interpretation: only code to model is incremental. | 19 |
| | | | $(\frac{1}{2}1^{-}\frac{1}{2})^{><}$ | HTML-MediaWiki (Ex.10): text updates in HTML side is prohibited, only HTML to MediaWiki generation is incremental. | 20 |
| | | | $(\frac{1}{2}1^{+}\frac{1}{2})^{>>}$ | Partial roundtripping (Ex.5) in the *-interpretation: only code generation is incremental. | 21 |
| | | | $(\frac{1}{2}1^{-}\frac{1}{2})^{>>}$ | HTML-MediaWiki (Ex.10): text updates in HTML side is prohibited, only MediaWiki to HTML generation is incremental. | 22 |
| | | 1 | $(\frac{1}{2}1^{+}1)$ | Partial roundtripping (Ex.5) in the *-interpretation: both directions are incremental. | 23 |
| | | | $(\frac{1}{2}1^{-}1)$ | HTML-MediaWiki (Ex.10): text updates in HTML side is prohibited, both direction is incremental. | 24 |
| Org = 1 | Inf = 0 | 0 | $(1^{+}00)$ | Roundtripping (Ex. 4): both directions are non-incremental. | 25 |
| | | | $(1^{-}00)$ | Bi-partial roundtripping (Ex. 8): both directions are non-incremental. | 26 |
| | | ½ | $(1^{+}0\frac{1}{2})^{><}$ | Roundtripping (Ex. 4): only code generation is incremental. | 27 |
| | | | $(1^{-}0\frac{1}{2})^{><}$ | Bi-partial roundtripping (Ex. 8): only code generation is incremental. | 28 |
| | | | $(1^{+}0\frac{1}{2})^{>>}$ | Roundtripping (Ex. 4): only incremental from code to model. | 29 |
| | | | $(1^{-}0\frac{1}{2})^{>>}$ | Bi-partial roundtripping (Ex. 8): only incremental from code to model. | 30 |
| | | 1 | $(1^{+}01)$ | Roundtripping (Ex. 4): both directions are incremental.<br>Outline view in JDT (Ex.6): the view is fully editable and changes are incremental in both directions. | 31 |
| | | | $(1^{-}01)$ | Bi-partial roundtripping (Ex. 8): both directions are incremental. | 32 |
| | Inf = 1 | 0 | $(1^{+}1^{+}0)$ | Roundtripping (Ex. 4) in the *-interpretation: both directions are non-incremental. | 33 |
| | | | $(1^{+}1^{-}0)$ | HTML-MediaWiki (Ex.10): both directions are non-incremental. | 34 |
| | | | $(1^{-}1^{+}0)$ | Bi-partial roundtripping (Ex. 8) in the *-interpretation: both directions are non-incremental. | 35 |
| | | | $(1^{-}1^{-}0)$ | HTML-MediaWiki (Ex.10): text updates and layout updates are prohibited in HTML and MediaWiki side, respectively, and both directions are non-incremental. | 36 |
| | | ½ | $(1^{+}1^{+}\frac{1}{2})$ | Roundtripping (Ex. 4) in the *-interpretation: only one direction is incremental. | 37 |
| | | | $(1^{+}1^{-}\frac{1}{2})$ | HTML-MediaWiki (Ex.10): only one direction is incremental. | 38 |
| | | | $(1^{-}1^{+}\frac{1}{2})$ | Bi-partial roundtripping (Ex. 8) in the *-interpretation: only one direction is incremental. | 39 |
| | | | $(1^{-}1^{-}\frac{1}{2})$ | HTML-MediaWiki (Ex.10): text updates and layout updates are prohibited in HTML and MediaWiki side, respectively, and only one direction is incremental. | 40 |
| | | 1 | $(1^{+}1^{+}1)$ | Roundtripping (Ex. 4) in the *-interpretation: both directions are incremental. | 41 |
| | | | $(1^{+}1^{-}1)$ | HTML-MediaWiki (Ex.10): both directions are incremental. | 42 |
| | | | $(1^{-}1^{+}1)$ | Bi-partial roundtripping (Ex. 8) in the *-interpretation: both directions are incremental.<br>Class and sequence diagrams (Ex. 9): both directions are incremental. | 43 |
| | | | $(1^{-}1^{-}1)$ | HTML-MediaWiki (Ex.10): text updates and layout updates are prohibited in HTML and MediaWiki side, respectively, and only both directions are incremental. | 44 |

Having classified scenarios, we can describe a model synchronization tool by the set of scenarios the tool supports. For example, unidirectional ATL (in its standard non-incremental version) supports synchronization of type (010) (with its info-rich and info-poor subtypes); GRoundTram Hidaka *et al.* (2011) is a tool for info-asymmetric bidirectional transformations that supports subtypes of type (101). QVT-R is intended to support bidirectional *rich* info-symmetric transformations ($11^{+}0$); however, semantic issues usually limit its application to the *poor* info-symmetric cases ($11^{-}0$).

## 2.4 Conclusion

The modern MDE requires a shift from model transformation pipelines to networks of interrelated models, and poses several challenges for maintaining the integrity of this network: support of bidirectionality, incrementality, informational symmetry, and ultimately concurrent updates create a package of non-trivial technological and theoretical issues to resolve. Having a taxonomy of synchronization behaviours, with a clear semantics for each taxonomic unit, can help to manage these problems. In this Chapter, we studied the various types of synchronizations scenarios and constructed a 3D taxonomic space that characterizes various synchronizations scenarios. In the presented taxonomic space, two dimensions are computational and form a plane classifying pairs of mutually inverse update propagation operations realizing a bidirectional transformation. The third dimension is orthogonal to the plane and classifies relationships of organizational dominance between the models to be kept in sync.

Our study shows that there exist 44 different concrete synchronization types, and this number might even increase when we add further dimensions to the taxonomy. The variety of the synchronization types implies the possible variations in the definition and the maintenance of their corresponding model transformations. Such variation besides the internal complexities of models would make the task of defining a transformation and maintaining the model interrelations complex. To deal with this complexity, we need to employ suitable abstraction techniques that offer declarative methods with clear semantics in transformation definitions. In the appendix of the published work by Diskin *et al.* (2016), the taxonomical space –presented in this chapter– is given formal semantics that is based on the QueST approach. The remaining chapters in this thesis show the concrete application of QueST in MT definitions and aim to make it accessible to the MDE community. The presented applications in this thesis are confined to the non-incremental, unidirectional, and informationally asymmetric scenarios according to the terminologies of this chapter.

# Chapter 3

# The Technical Framework

In this Chapter, we first define models, metamodels, and other concepts we will use. Then, we define high-level query operations that are used in the definition of MTs in QueST. The definitions in this Chapter are aimed to be formal, and can be skipped by the readers interested to know about the QueST approach in action; they can go directly to Chapter 4.

## 3.1 Diagrammatic Constraints

The definitions in this section are adapted from Rutle (2010) and Diskin and Wolter (2008). The machinery introduced in this chapter is called Diagrammatic Predicate Framework (DPF) by Rutle (2010). These definitions are necessary for the establishment of the definitions in Sect. 3.2 that are introduced in the current thesis. We assume the reader is already familiar with the model and metamodeling concepts in general. Although we have provided the necessary definition in this section, the reader might see the above two references for further examples and explanations about DPF.

### 3.1.1 Metamodels

We first define graph and graph homomorphism.

**Definition 1** (Graph). *A graph $g$ is a tuple $g = (V, E, s, t)$ where $V$ and $E$ are sets of nodes and edges, respectively; $s, t : E \to V$ are functions mapping each edge to its source and target node, respectively.*

**Definition 2** (Graph Homomorphism). *Let $g_1$ and $g_2$ be two graphs with $g_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$. A graph homomorphism $f : g_1 \to g_2$, $f = (f_V, f_E)$ consist of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions; i.e., $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.*

A graph homomorphism $f : g_1 \to g_2$ is inclusion if both $f_V$, $f_E$ are inclusions. We will denote inclusion homomorphisms with hook arrows (e.g., $f : g_1 \hookrightarrow g_2$).

A *Metamodel* is a graph with a (possibly empty) set of constraints (see Definition 5). If a metamodel does not have any constraints, a *model* of a metamodel is any other graph with a typing mapping (i.e., a graph homomorphism) targeting the metamodel graph (see Definition 8). However, usually metamodels have some constraints. Thus, we first need to define constraints.

Constraints are defined independently from any individual metamodel (see Rutle (2010), Table 3.1). They are defined as predicates whose input arities (i.e., parameters) have graph shape rather than being tuples. This the reason they are called diagrammatic constraints. We syntactically define the set of constraint (independently of any metamodel) as follows.

**Definition 3** (Signature of Diagrammatic Constraints)**.** *A signature of diagrammatic constraints* $\Sigma = (C^{\Sigma}, \alpha^{\Sigma})$ *consists of a set of constraint symbols* $C^{\Sigma}$ *and a mapping* $\alpha^{\Sigma}$ *that assigns an arity graph* $g$ *to each constraint symbol* $c \in C^{\Sigma}$*.*

We will refer to each $c \in C^{\Sigma}$ in the above definition as a $\mathscr{D}$*constraint*. For example, we might have a $\mathscr{D}$*constraint* $[1..*]$ whose arity graph is Fig. 3.1(a). Another example could be $[\subseteq]$ whose arity graph is Fig. 3.1(b). In practice, the signature of the diagrammatic constraints are defined in the modeling environment.

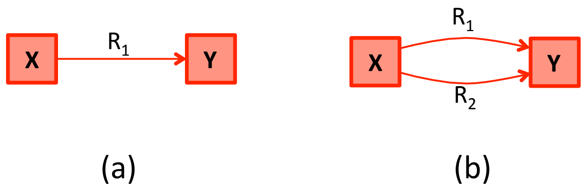


Figure 3.1: Arities of $\mathscr{D}$*constraints*

As mentioned before, a metamodel without any constraint is a simple graph. The definition of a $\mathscr{D}$*constraint* over a metamodel is a graph homomorphism from the arity of the $\mathscr{D}$*constraint* to the metamodel graph. We call this a marking of the metamodel graph with a $\mathscr{D}$*constraint*.

Note that constraints are applied to a graph shape. Thus, there might be constraints whose arity is a single node, or whose arity is a single arrow, or whose arity is an arbitrary finite graph. The first two types are just special cases of the general case.

**Definition 4** (Marking of Metamodel with $\mathscr{D}$*constraint*)**.** *A marking* $M_c$ *of a (metamodel) graph* $G_{\mathbb{M}}$ *with a* $\mathscr{D}$*constraint* $c$ *is defined as a graph homomorphism from* $\alpha^{\Sigma}(c)$ *to* $G_{\mathbb{M}}$*.*

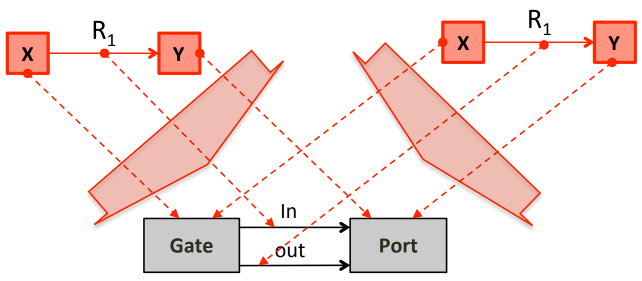


Figure 3.2: Explicit definition of two atomic constraints.

Each marking of a metamodel is called an *atomic constraint* or an *instance* of a $\mathscr{D}$*constraint*. For example, Fig. 3.2 depicts two atomic constraint that are instances of the same $\mathscr{D}$*constraint* [1..0].

We are ready to define metamodel as follows.

**Definition 5** (Metamodel)**.** *A* metamodel $\mathbb{M}$ *is a graph* $G_{\mathbb{M}}$ *that is equipped with a (possibly empty) set of markings with* $\mathscr{D}$*constraints taken from a predefined signature of diagrammatic constraints.*

## 3.1.2 Models

If a metamodel $\mathbb{M}$ is not being marked with any $\mathscr{D}$*constraint* (i.e., does not have any atomic constraints), any other graph typed over $\mathbb{M}$ is a valid model for the metamodel. The atomic constraints restrict the space of valid models to those that are *compatible* with the constraints. To make these statements precise, we first need to define the *semantics* of each $\mathscr{D}$*constraint*; then, we define the *compatibility* notion.

**Definition 6** (Semantics of $\mathscr{D}$*constraints*)**.** *A semantic interpretation* $[\![..]\!]^{\Sigma}$ *of a signature of diagrammatic constraints* $\Sigma = (C^{\Sigma}, \alpha^{\Sigma})$ *is given by a mapping that assigns to each* $\mathscr{D}$*constraint* $c \in C^{\Sigma}$ *a set* $[\![c]\!]^{\Sigma}$ *of graph homomorphisms* $i : O \to \alpha^{\Sigma}(c)$ *that is called valid instances of* $c$*.*

A modeling framework might employ different approaches in practice to specify $\mathscr{D}$*constraint* semantics. One approach is to assign a boolean function to each $\mathscr{D}$*constraint* $c$ that takes a graph homomorphism $i : O \to \alpha^{\Sigma}(c)$ as input and returns either true or false, depending on the $\mathscr{D}$*constraint* intended semantics. Another approach is to consider the converse of the homomorphism mappings, and assume that nodes in the arity graph (of $\mathscr{D}$*constraints*) are interpreted as sets and arrows are interpreted as relations (see Sect. 4.1.2) and use concise relational notation to specify the semantics (see Table 3.1 in Rutle (2010)). Thus, the language used to specify the semantics of the constraints is a parameter to the DPF framework.

Models of a metamodel should be compatible with the atomic constraints. To define the *compatibility* notion precisely, we resort to the pullback operation in the category of graphs and homomorphisms. For further information about the Pull Back (PB) operation in this Category see the book by Ehrig *et al.* (2006).

**Definition 7** (Constraint Compatibility)**.** *A typing mapping $i : O \to G_{\mathbb{M}}$ is compatible with a marking (i.e., atomic constraint) $M_c : c \to G_{\mathbb{M}}$ on a metamodel $\mathbb{M}$ if $i'$ in the following pullback diagram is an element of $[\![c]\!]^\Sigma$ (i.e., $i' \in [\![c]\!]^\Sigma$).*

$$\begin{tikzcd}
G_{\mathbb{M}} & c \arrow[l, "M_c"'] \\
O \arrow[u, "i"] & O' \arrow[l, "M'_c"] \arrow[u, "i'"']
\end{tikzcd}$$

To check the compatibility of the constraint $c$, we only care about the parts of the typing mapping whose image is the same as the image of the $c$ marking (i.e., image of $M_c$). Thus, according to the above definition, we extract this part (i.e., $O' \to c$) and check whether it is a valid homomorphism with respect to the $c$ semantics.

We now define a model of the metamodel as a typing mapping that conforms to all the atomic constraints defined over the metamodel graph, as follows.

**Definition 8** (Model)**.** *A model $M$ conforming to a metamodel $\mathbb{M}$ is a pair $M = (O, T_m)$ such that $O$ is a graph, and $T_m : O \to G_{\mathbb{M}}$ is a graph homomorphism that is compatible with all the markings (i.e., atomic constraints) on $\mathbb{M}$.*

Morphism $T_m : O \to G_{\mathbb{M}}$ is sometimes called a pre-instance of the metamodel $\mathbb{M}$. If a pre-instance is compatible with all the atomic constraints of $\mathbb{M}$, it is called a (valid) model of $\mathbb{M}$.

Definition 5 does not include the possible attributes[1] that might be associated to nodes and arrows in a metamodel. Accordingly, Definition 8 does not capture the semantics (i.e., values) of such attributes. A more detailed formalization of models and metamodels might be considering metamodels as attributed graphs, and models as typed attributed graphs (see Ehrig *et al.* (2006) for the definition of these terms). The syntax and semantics of the constraints defined in Definition 3 and Definition 6 in this thesis are, accordingly, based on this simpler interpretation of models and metamodels. Thus, they are not intended to capture the constraints defined over the attributes of the nodes and arrows in metamodels. Given these consideration and owing to the assumption that models are considered finite graphs in MDE, the verification of the constraints compatibility in Definition 7 is always decidable.

[1]By attributes, we almost mean what is called properties in UML

## 3.2 Diagrammatic Queries

Query operations are the main building blocks of MT definitions in QueST (see Chapter 4). The queries in QueST are defined over metamodels and are executed over models. The input and output parameters in these queries are collections and the shape of these parameters are graphical. This is the reason we call them diagrammatic queries ($\mathscr{D}$*query*s). In this section, we answer questions such as: What is the $\mathscr{D}$*query* syntax? How is the semantics of a $\mathscr{D}$*query* defined? How is a $\mathscr{D}$*query* defined over a metamodel? Finally, how is it executed over a model? We also show that the definition of $\mathscr{D}$*query* execution is compatible with the query concepts in the sense that the execution of $\mathscr{D}$*query*s leave the original model elements intact.

### 3.2.1 Query Syntax

Each $\mathscr{D}$*query* is syntactically identified with a query symbol and its corresponding input/output arity that is a graph homomorphism defined as follows.

**Definition 9** (Diagrammatic Arity)**.** *A diagrammatic arity $\mathscr{A} : A^{in} \hookrightarrow A^{out}$ is an inclusion graph homomorphism with a domain $A^{in}$ and a codomain $A^{out}$. $A^{in}$ and $A^{out}$ are called the input and the output arity, respectively.*

Each modeling framework defines a set of $\mathscr{D}$*query*s (i.e., the query language signature). The users use these $\mathscr{D}$*query*s to define MT definitions in QueST.

**Definition 10** (Signature of Diagrammatic Queries)**.** *A signature of diagrammatic queries $\mathscr{D} = (\mathcal{Q}^{\mathscr{D}}, \alpha^{\mathscr{D}})$ consists of a set of $\mathscr{D}$query symbols $\mathcal{Q}^{\mathscr{D}}$ and a mapping $\alpha^{\mathscr{D}}$ that assigns a diagrammatic arity $\mathscr{A}$ to each $\mathscr{D}$query symbol $Q \in \mathcal{Q}^{\mathscr{D}}$.*

For example, we can define a signature that includes three query operations $Q_1$, $Q_2$, $Q_3$, whose corresponding diagrammatic arities are depicted in Fig. 4.15, Fig. 4.16 and Fig. 4.17, respectively.

As seen above, a $\mathscr{D}$*query* signature (and also its semantics – as defined in Definition 14) is defined independently of any metamodel. To define a $\mathscr{D}$*query* over a metamodel, the user marks the metamodel graph with the $\mathscr{D}$*query*'s corresponding input arity. Such a marking is a graph homomorphism from the input arity of the $\mathscr{D}$*query* into the metamodel graph as follows.

**Definition 11** (Marking of Metamodel with $\mathscr{D}$*query*)**.** *Let $Q^{in}$ be the input arity of a $\mathscr{D}$query $Q$. A marking $m$ of a (metamodel) graph $G_{\mathbb{M}}$ with the $\mathscr{D}$query $Q$ is a graph homomorphism $m : Q^{in} \to G_{\mathbb{M}}$.*

### Metamodel Augmentation

Whenever a metamodel is marked with the input arity of a $\mathscr{D}query$ $Q$, the QueST framework will augment the metamodel –by adding new nodes and arrows– according to the $Q$ output arity. The augmentation process is an application of a graph transformation rule defined as follows.

**Definition 12** (Metamodel Augmentation). *An augmentation of a metamodel graph $G_{\mathbb{M}}$ with a $\mathscr{D}query$ marking $m : Q^{in} \to G_{\mathbb{M}}$ is defined by the following pushout diagram:*

$$\begin{tikzcd}
Q^{in} \arrow[r, hook, "i"] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & Q^{out} \arrow[d, "m'"] \\
G_{\mathbb{M}} \arrow[r, hook, "i'"'] & Q(G_{\mathbb{M}})
\end{tikzcd}$$

In the above diagram, $Q(G_{\mathbb{M}})$ is called the augmented metamodel (graph). Note that $i'$ is an inclusion mapping, since $i$ is so[1]. This means that the original elements in the metamodel are kept intact during the augmentation process. $m'$ associates the newly generated elements on the augmented metamodel with their corresponding elements in the $\mathscr{D}query$ output arity. For the concrete examples of augmentation process, see Sect. 4.4.

### 3.2.2 $\mathscr{D}query$ Semantics

We consider the input values of each $\mathscr{D}query$ to be homomorphisms into the input arity of $\mathscr{D}query$. So, we first define the collection of such homomorphisms as follows.

**Definition 13** (Model Space). *The model space $\mathbb{G}$ of a graph $G$ is the collection of all homomorphisms (i.e., models) $t : V \to G$, such that $V$ is just another graph.*

Let $Q^{in}$ be the input arity of a $\mathscr{D}query$ $Q$. For a given input value (e.g., $t_1 : V \to Q^{in}$, $t_1 \in \mathbb{Q}^{in}$), the semantics of $Q$ should specify the way $t_1$ will be augmented: which elements will be added to $V$, and over which elements in $Q^{out} - Q^{in}$ these new elements will be typed. The constraint over the $\mathscr{D}query$ semantic specification is that they should build pullback diagrams as follows.

**Definition 14** (Semantics of $\mathscr{D}query$s). *Let $\mathbb{Q}^{in}$ be the model space of the input arity of the query $Q$. A semantic interpretation $[\![..]\!]^{\mathscr{D}}$ of a diagrammatic operation signature*

[1]In a general category, pushouts do not necessarily preserve monomorphisms. However, this holds true in the category of graphs and graph homomorphisms (see Ehrig *et al.* (2006), Fact 2.17)

*$\mathscr{D} = (\mathscr{Q}^{\mathscr{D}}, \alpha^{\mathscr{D}})$ is given by a mapping that assigns to each $\mathscr{D}$query $Q \in \mathscr{Q}^{\mathscr{D}}$ a function $[\![Q]\!]^{\mathscr{D}}$ that accepts a model $m_{in} : V \to Q^{in}$ from the model space $\mathbb{Q}^{in}$ as an input, and returns a pullback diagram as follows:*

$$\begin{tikzcd}
Q^{in} \arrow[r, hook, "i"] & Q^{out} \\
V \arrow[u, "m_{in}"] \arrow[r, hook, "i'"'] & V' \arrow[u, "m_{out}"']
\end{tikzcd}$$

In the above definition, $i$ and $i'$ are inclusion homomorphisms and $m_{in}$ and $m_{out}$ are models in the model spaces of $Q^{in}$ and $Q^{out}$, respectively. Note that the semantic function gets a model and returns a diagram. However, it is sometimes easier to think it gets as input a model $m_{in} : V \to Q^{in}$ and returns as output a model $m_{out} : V' \to Q^{out}$, and implicitly consider the existence of the above pullback diagram. The reason we require the semantic function to return a pullback diagram is because it ensures that no new data is added to the original types in $Q^{in}$. This is a fundamental requirement that distinguishes queries from updates.

We assume that all our queries are computable; that is, there is an algorithm that takes a model (i.e., a typing mapping) as an input, and return a pullback square of mappings as the output as shown above.

## Query Execution

Let $Q^{in} \hookrightarrow Q^{out}$ be the diagrammatic arity of a $\mathscr{D}$*query* $Q$, and assume the marking $Q^{in} \to G_{\mathbb{M}}$ is defined over the metamodel graph $G_{\mathbb{M}}$. Thus, we will have the following diagram (after the execution of the metamodel augmentation step – see metamodel augmentation in Sect. 3.2.1):

$$\begin{tikzcd}
Q^{in} \arrow[r, hook] \arrow[dr, "m"] & Q^{out} \arrow[dr, "m'"] \arrow[d, phantom, "\text{PO}"] & \\
 & G_{\mathbb{M}} \arrow[r, hook] & Q(G_{\mathbb{M}})
\end{tikzcd}$$

Let also $t : O \to Q^{in}$ be a valid model of the metamodel $\mathbb{M}$. Thus we will have the following diagram.

$$\begin{tikzcd}
Q^{in} \arrow[rr, hook, "i"] \arrow[dr, "m"] & & Q^{out} \arrow[dr, "m'"] & \\
& G_{\mathbb{M}} \arrow[rr, hook, "q"] & & Q(G_{\mathbb{M}}) \\
& O \arrow[u, "t"'] & &
\end{tikzcd}$$

We also assume that $[\![Q]\!]$ (i.e., the semantics of $Q$) is provided according to Definition 14. The QueST framework proceeds with the execution of $Q$ as follows:

1. Using the marking $m$ and the typing mapping $t$, the framework executes the pullback operation on the diagram $Q^{in} \overset{m}{\hookrightarrow} G_{\mathbb{M}} \overset{t}{\leftarrow} O$ as follows.

$$\begin{tikzcd}
Q^{in} \arrow[rr, hook, "i"] \arrow[dr, "m"] & & Q^{out} \arrow[dr, "m'"] & \\
& G_{\mathbb{M}} \arrow[rr, hook, "q"] & & Q(G_{\mathbb{M}}) \\
O_{*} \arrow[uu, "b"] \arrow[dr, "j"] \arrow[ur, phantom, "\text{PB}"] & & & \\
& O \arrow[uu, "t"'] & &
\end{tikzcd}$$

The pullback execution generates $Q^{in} \overset{b}{\leftarrow} O_{*} \overset{j}{\rightarrow} O$ from which the homomorphism $Q^{in} \overset{b}{\leftarrow} O_{*}$ is the value homomorphism for the $Q$ operation. (It is, in fact, a model within the $Q^{in}$ model space.)

2. Using $[\![Q]\!]$, the framework completes the diagram in the previous step to the one below. Note that the completed back face in a pullback diagram according to Definition 14.

$$\begin{tikzcd}
Q^{in} \arrow[rr, hook, "i"] \arrow[dr, "m"] & & Q^{out} \arrow[dr, "m'"] & \\
& G_{\mathbb{M}} \arrow[rr, hook, "q"] & & Q(G_{\mathbb{M}}) \\
O_{*} \arrow[uu, "b"] \arrow[rr, hook, "i'"] \arrow[dr, "j"] & & O'_{*} \arrow[uu, "b'"'] & \\
& O \arrow[uu, "t"'] & &
\end{tikzcd}$$

3. The framework executes the pushout operation on the bottom face (i.e., $O \leftarrow O_* \hookrightarrow O'_*$) in the above diagram and brings back the results of the query execution to the model as follows.

$$\begin{tikzcd}
Q^{in} \arrow[rr, hook, "i"] \arrow[dr, "m"] & & Q^{out} \arrow[dr, "m'"] & \\
& G_{\mathbb{M}} \arrow[rr, hook, "q"] & & Q(G_{\mathbb{M}}) \\
O_* \arrow[uu, "b"] \arrow[rr, hook, "i'"] \arrow[dr, "j"] & & O'_* \arrow[uu, "b'"] \arrow[dr, blue] & \\
& O \arrow[uu, "t"] \arrow[rr, hook, blue, "i''"] \arrow[ur, phantom, "\text{PO}"] & & {\color{blue}O'}
\end{tikzcd}$$

4. Finally, it draws a homomorphism $t' : O' \rightarrow Q(G_{\mathbb{M}})$ that completes the picture to the one depicted in Fig. 3.3. This homomorphism is uniquely identified as follows. Since the following diagram (i.e., the bottom face in the cube) is a pushout diagram,

$$\begin{tikzcd}
O_* \arrow[r, hook, "i'"] \arrow[d, "j"] & O'_* \arrow[d, blue, "j'"'] \\
O \arrow[r, hook, blue, "i''"'] & {\color{blue}O'}
\end{tikzcd}$$

for any other object K as shown in the diagram bellow with homomorphisms $x : O \rightarrow K$ and $z : O'_* \rightarrow K$ for which the following diagram commutes, there must exist a unique homomorphism $t'$ from $O'$ to $K$ also making the diagram commute:

$$\begin{tikzcd}
O_* \arrow[r, hook, "i'"] \arrow[d, "j"] & O'_* \arrow[d, blue, "j'"'] \arrow[ddr, bend left, "z"] & \\
O \arrow[r, hook, blue, "i''"'] \arrow[drr, bend right, "x"] & {\color{blue}O'} \arrow[dr, dashed, "t'"] & \\
& & K
\end{tikzcd}$$

Let us take $x$, $z$, and $K$ as the following: $x = t \circ q$, $z = b' \circ m'$, and $K = Q(G_{\mathbb{M}})$. Since $j \circ x = i' \circ z$ (because of the commutativity of the diagrams in the other surfaces), there exists a unique homomorphism from $O'$ to $Q(G_{\mathbb{M}})$.

$$\begin{tikzcd}
Q^{in} \arrow[rr, hook, "i"] \arrow[dr, "m"] & & Q^{out} \arrow[dr, "m'"] & \\
& G_{\mathbb{M}} \arrow[rr, hook, "q"] & & Q(G_{\mathbb{M}}) \\
O_* \arrow[uu, "b"] \arrow[rr, hook, "i'"] \arrow[dr, "j"] & & O'_* \arrow[uu, "b'"] \arrow[dr] & \\
& O \arrow[uu, "t"] \arrow[rr, hook, "i''"] & & O' \arrow[uu, blue, "t'"]
\end{tikzcd}$$

Figure 3.3: A $\mathscr{D}query$ Execution cube

An interesting fact about the diagram in Fig. 3.3 is that the front and the right faces are also pullback diagrams as shown by the following lemma.

**Lemma 1.** *In the diagram at Fig. 3.3 the front and right faces are pullbacks.*

*Proof.* The category of graphs and homomorphisms is an Adhesive Category, and according to the Adhesive Category definition, pushouts along mono morphisms are VK squares (Ehrig *et al.*, 2006). The VK square definition is as follows. In Fig. 3.4, a pushout (1) is a VK square if, for any commutative cube (2) with (1) in the bottom and where the back faces are pullbacks, the following statement holds: the top face is a pushout iff the front faces are pullbacks (we have borrowed the definition from Ehrig *et al.* (2006)).

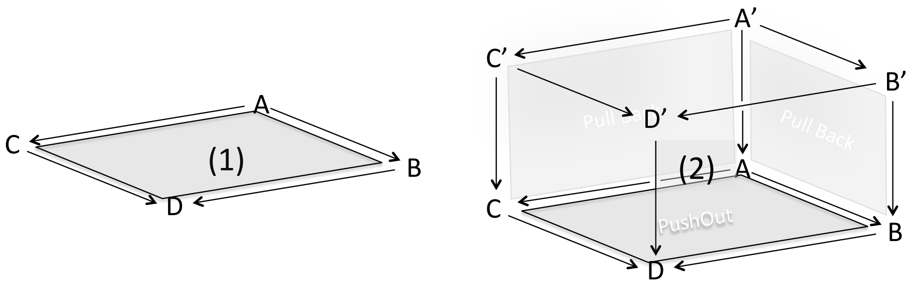


Figure 3.4: Van Kampen square

Thus, according to the above definition, since in Fig. 3.3 the top face is a pushout along a monomorphism, and the left and the back faces are pullbacks, and further the bottom face is a pushout, then the right and the front faces must be pullbacks. □

The front face being a pullback in Fig. 3.3 clarifies that a $\mathscr{D}query$ execution does not add any new data to the original types in the metamodel, as is expected. The right face being pullback ensures that any new data (that is typed over the image of $m'$) in the model is exactly the data generated by the $\mathscr{D}query$.

The VK diagram in Fig. 3.3 is created as a result of one query execution. Lets call it $VK_1$. The next query execution would create another VK diagram $VK_2$. The front right edge of $VK_1$ (i.e., $t'$ in Fig. 3.3) will be the front left edge of $VK_2$. In a similar way, the third query execution making $VK_3$ will have a shared edge with $VK_2$.

## 3.3 Conclusion

This chapter provides the basic definitions necessary for the introduction of the QueST approach. Metamodels are defined to be diagrammatic specifications –i.e., graphs with diagrammatic constraints ($\mathscr{D}constraint$s). Diagrammatic constraints are defined as predicates whose input parameters are of graph shape rather than tuples. They are defined independently of any metamodel and are attached to a metamodel during its definition. Models of a metamodel are defined to be homomorphisms into the metamodel graph that are compatible with its constraints.

This chapter also defines high-level diagrammatic query operations ($\mathscr{D}query$s). Each $\mathscr{D}query$ has a graph shape input/output arity. Similarly to $\mathscr{D}constraint$s, $\mathscr{D}query$s are defined independently of metamodels or any MT definition. An application of a $\mathscr{D}query$ over a metamodel is defined by a homomorphism from its input arity into the metamodel graph. The chapter defines the constraints for the $\mathscr{D}query$ semantic definitions and precisely explains each $\mathscr{D}query$ execution over a provided model based on its provided semantic definitions. The chapter also shows that the semantic definition of $\mathscr{D}query$s are consistent with the general functionality of queries as they keep the original elements intact by building pullback diagrams. Examples of DQF queries applications will be provided in Chapter 4 when we explain the QueST approach. All the model and metamodel graphs considered in the thesis are assumed to be finite. All the $\mathscr{D}constraint$s considered in this thesis are assumed to be decidable, and all the $\mathscr{D}query$ operations are computable.

# Chapter 4

# Model Transformation in QueST

"The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."

Edsger Dijkstra

In this chapter, we explain the QueST approach to the definition and the execution of model transformations. As mentioned earlier in Chapter 1, the transformation definition in QueST is similar to the definition of the views in relational databases; that is, a target model is defined to be a view over the source model. The view specification is defined via a series of queries defined over the source metamodel. Accordingly, an MT execution happens by executing these queries over the provided source model and generating the target model elements in a straightforward relabelling process. We will start our explanation of the above concepts by introducing a simple example and applying the QueST approach to it.

## 4.1 The HappyPeople Example

In QueST, a model transformation is defined on the metamodel layer and is executed on the model layer. Thus, we first introduce the metamodels that are involved in the running example.

### 4.1.1 Metamodels

In Chapter 3, we defined metamodels to be a graph with a (possibly empty) set of diagrammatic constraints. We might refer to the nodes in a metamodel as either classes or types, and edges as either arrows or associations. Fig. 4.1 shows the source

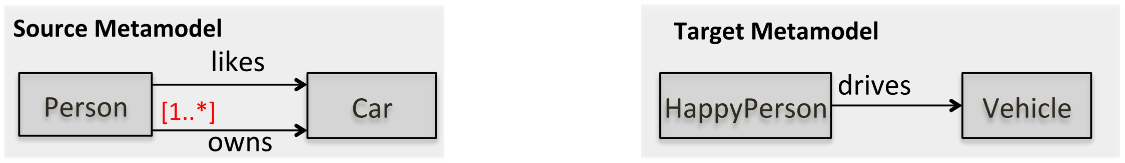


Figure 4.1: The HappyPeople example metamodels

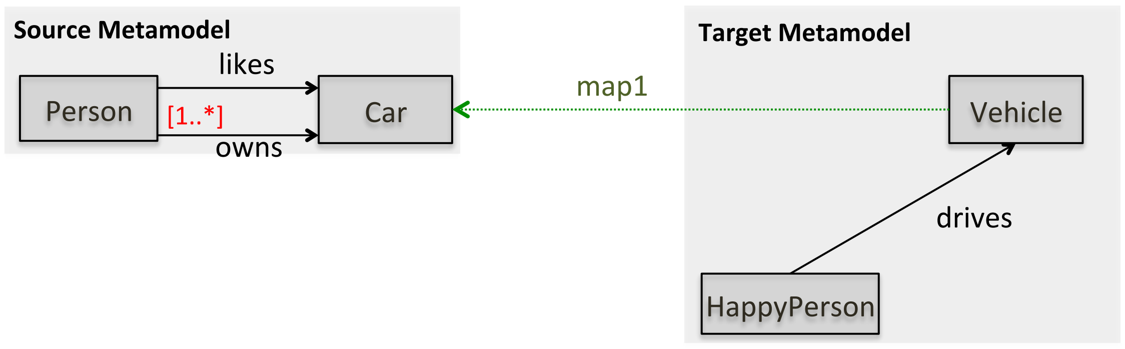


Figure 4.2: QueST definition of the HappyPeople MT

metamodel on the left and the target metamodel on the right. The source metamodel has two nodes, `Person` and `Car`, and two arrows, `likes` and `owns` between `Person` and `Car` indicating whether or not a person likes or owns a car in a provided model. The [1..*] multiplicity constraint on `owns` denotes that no car is left without an owner in a corresponding model. This can be precisely specified as $\forall c : Car, \exists p : Person | (p, c) \in owns$. The target metamodel also has two nodes, `HappyPerson` and `Vehicle`, but there exists only one arrow called `drives` between the latter two nodes denoting whether or not a (happy) person drives a vehicle.

### 4.1.2 Metamodel's Set/Relation Interpretation

In Chapter 3, we defined the models of a metamodel to be a family of homomorphism into to the metamodel graph that are compatible with the metamodel constraints (see Definition 8). In a model $t : O \to \mathbb{M}$, the graph $O$ is called the datagraph of the model. We refer to the nodes and edges of a datagraph as objects and references, respectively. For example, Fig. 4.3 illustrates a model of the source metamodel in Fig. 4.2. This datagraph has eight nodes: $p_i, i = 1..3$ typed over `Person`, and $c_i, i = 1..3$ typed over `Car`. It has seven references: $l_i, i = 1..4$ typed over `likes`, and $o_i, i = 1..3$ typed over `owns`. The colon notation in this figure is used as an abbreviation to typing mapping links (e.g., $l_3$:`likes` means $l_3$ is typed over `likes`).

The directions of the typing mapping links —as seen in Fig. 4.3— are from the

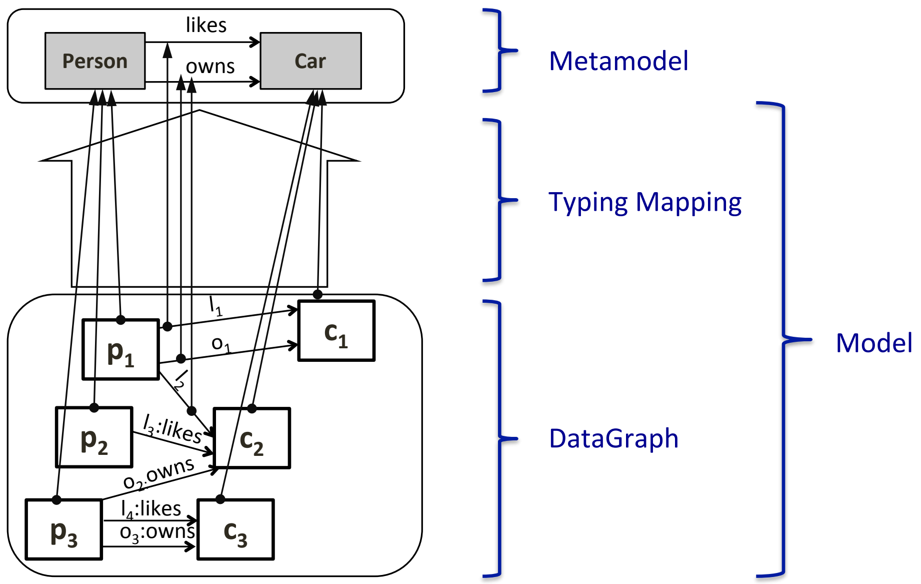


Figure 4.3: Model of a metamodel as a typed graph

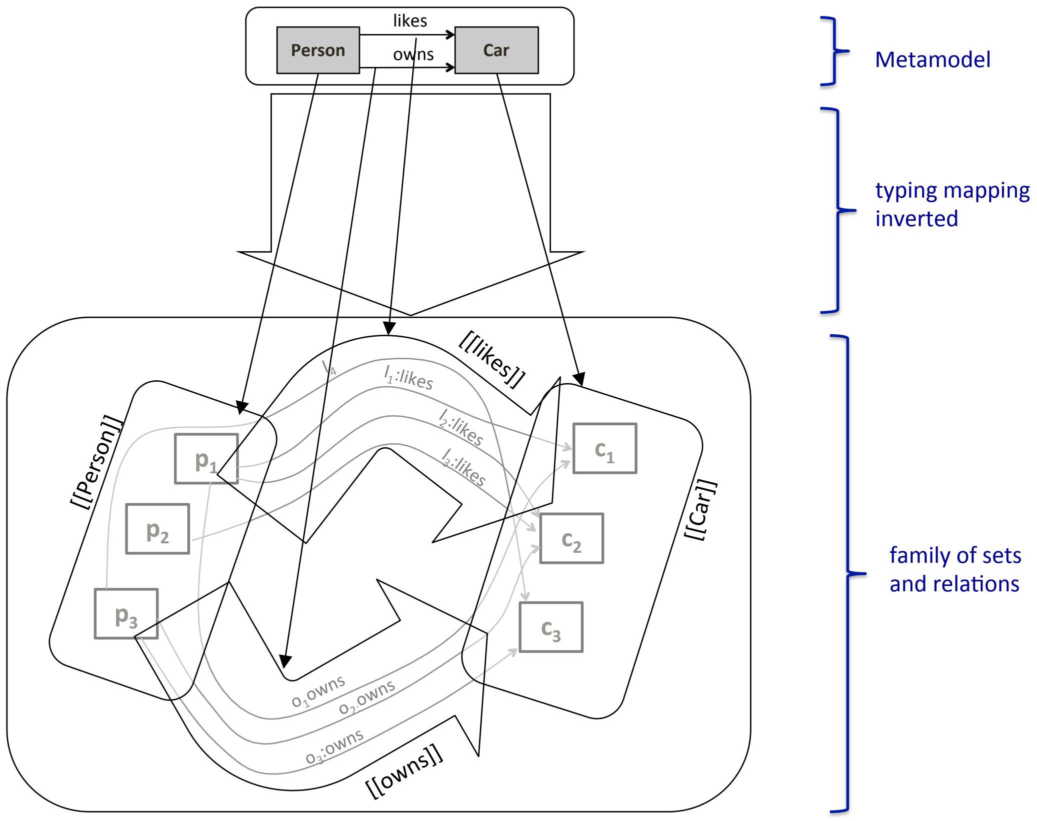


Figure 4.4: Typing mapping inverted

datagraph to the metamodel graph. However, it is sometimes convenient to work with the converse of these links; that is, the converse links connect each class to its corresponding objects, and each association to its corresponding references in the datagraph. Fig. 4.4 shows the same model as Fig. 4.3 at which the typing mapping links are inverted. By inverting typing mapping links, `Car` is now mapped to three elements $c_i, i = 1..3$, and `Person` is mapped to three elements $p_i, i = 1..3$. Thus, we can interpret `Car` and `Person` as follows:

- $[\![Car]\!] = \{c_i\}, i = 1..3$
- $[\![Person]\!] = \{p_i\}, i = 1..3$[1].

The story of arrows is different. Two arrows in Fig. 4.4 are mapped as follows: the arrow `likes` is mapped to references $l_i, i = 1..4$ and the arrow `owns` is mapped to references $o_i, i = 1..3$. Thus, they will be interpreted as follows:

- $[\![likes]\!] = \{l_i\}, i = 1..4$
- $[\![owns]\!] = \{o_i\}, i = 1..3$

The elements of the above sets are edges in the data graph. Each edge has a source and a target node. If we replace each edge with its corresponding pair of source and target nodes, we will have the following sets of pairs.

- $[\![likes]\!] = \{(p_1, c_1), (p_1, c_2), (p_2, c_2), (p_3, c_3)\}$
- $[\![owns]\!] = \{(p_1, c_1), (p_3, c_2), (p_3, c_3)\}$

In replacing the edges with their corresponding pairs, it is possible we end up having duplicates in either $[\![likes]\!]$ or $[\![owns]\!]$. The reason is that nothing in the metamodel definition (see Definition 8) prevents the two distinct edges having the same source and target objects being typed over the same arrow in the metamodel. If we add a constraint to our metamodel definition to prevent such cases[2], we can avoid duplicates in the above collections. By imposing this restriction, it is easy to see that the above pairs are occurrences of the two following relation declarations.

- $likes \subseteq Car \times Person$
- $owns \subseteq Car \times Person$

**Remark 1.** *In this thesis, we assume that two distinct references whose source and target objects are the same in the datagraph can not be typed over the same arrow in the metamodel graph.*

[1]Instead of $[\![Car]\!] = \{c_i\}$ notation, we might simply write $Car = \{c_i\}$.

[2]UML declares such a constraint by declaring the respective association end to be unique.

### 4.1.3 MT Specification

We intend to define a simple transformation between the models of the metamodels in Fig. 4.1. The transformation specification is to translate models specifying people who own and like cars to models of happy people who drive vehicles. There are two requirements for the translation. The first is that cars and vehicles are considered to be synonymous: every car in a source model should be a vehicle in the target model, and vice versa. The second is a (very modest) criterion of happiness: A person in the source is considered to be happy if there is at least one car that he both `likes` and `owns`, and then he is allowed to drive such a car (or cars). In the next section, we will see how we can define this transformation in QueST.

## 4.2 MT Definition in QueST

The target metamodel structure acts as a guideline in proceeding with the transformation definition in QueST; that is, for each element of the target, we need to identify its corresponding element in the source, and if such an element does not exist, we define it via queries. Thus, each class or association type in the target metamodel should be mapped to a class or association type either originally present in the source metamodel or defined by queries. Semantically, the mappings specify how the instances of the target types are generated: if type X in the target is mapped to type Y in the source, for every instance of X, an instance of Y will be generated during the transformation execution[1]. We will define this execution semantics more precisely in Sect. 4.3.1.

The user defines the target-to-source mappings according to the provided informal MT specification. For example, in the running example, class `Vehicle` is mapped to class `Car` (see *map1* in Fig. 4.2), since there should be a one-to-one correspondence between them, given the requirements of this MT in Sect. 4.1.3. For the other elements in the target, we do not have corresponding direct elements in the source, so we need to define the required elements by applying queries.

### 4.2.1 Metamodel Augmentation

Many target types do not usually have corresponding direct types among the source metamodel elements; for instance, according to the HappyPeople specification, the source metamodel does not have types directly corresponding to `HappyPerson` and `drives` from the target metamodel. The QueST idea is to make explicit such "missing" elements by finding suitable queries against the source metamodel. Application

[1]Note that though the transformation will be executed in the source-to-target direction, the mappings go in the opposite direction.

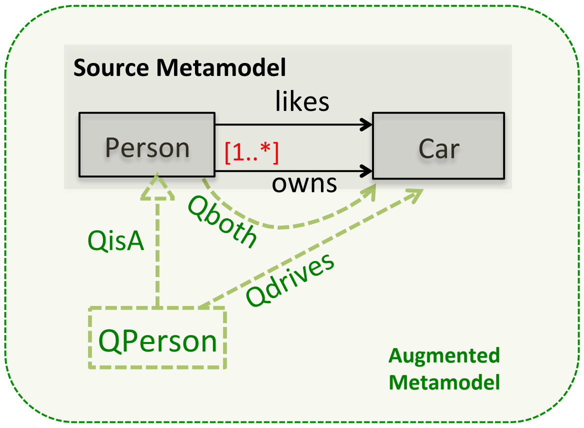


Figure 4.5: Metamodel Augmentation

of such queries will allow us to define derived elements on the source side, and complete the mapping by linking `HappyPerson` and `drives` with these derived elements. We define the following three queries, namely, $Q_1$, $Q_2$, and $Q_3$ to augment the source metamodel:

1. $Q_1$: ***Relation Intersection.*** This query takes the relation symbols `owns` and `likes`, and produces another relation symbol `Qboth` that semantically is their intersection relation (note the dashed curved green arrow labeled `Qboth` in Fig. 4.5).

2. $Q_2$: ***Domain Selection.*** This query takes the relation symbol `Qboth` as an input and produces a sort symbol `QPerson` and a relation symbol `QisA` from `QPerson` to `Person`. Semantically, `QPerson` is a subset of `Person` for which the relation `Qboth` is defined and `QisA` is a subsetting relation (note the `QPerson` dashed block square and the hollow-ended dashed green arrow `QisA` from `QPerson` to `Person` denoting a subsetting relation in Fig. 4.5).

3. $Q_3$: ***Arrow Composition.*** This query takes the relation symbols `Qboth` and `QisA` as inputs and produces another relation symbol called `Qdrives`. Semantically, `Qdrives` is the composition of `QisA` and `Qboth` (see the dashed green arrow labeled `Qdrives` between `QPerson` and `Car`.) In fact, `Qdrives` = `Qboth` as a set of pairs, but their domains are different.

As depicted in Fig. 4.5, definitions of the queries *augment* the source metamodel by introducing new classes and associations. These augmented elements are all shown with dashed green borders. The names for the derived elements are all prefixed with the letter ‘Q’ to distinguish them from the original metamodel elements.

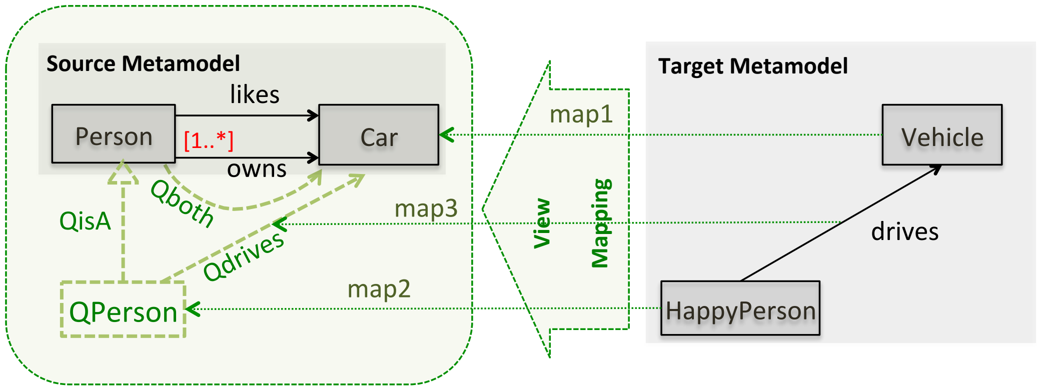


Figure 4.6: QueST definition of the HappyPeople MT

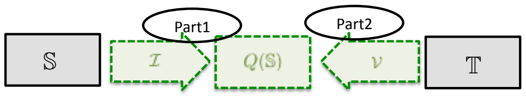


Figure 4.7: An MT definition abstract view in QueST

### 4.2.2 View-mapping Completion

After the source metamodel is appropriately augmented by the queries, we can complete the MT definition by mapping the remaining unmapped elements in the target. As shown in Fig. 4.6, *map2* and *map3* complete the target-to-source mapping in our example. All the target-to-source mappings[1] (i.e., *map1*, *map2* and *map3*) are collectively encapsulated within the block arrow called *view mapping* in this figure.

Fig. 4.7 abstracts away some details from the Fig. 4.6 and abstractly depicts a declarative MT definition structure in QueST. $\mathbb{S}$ and $\mathbb{T}$ are the source and target metamodels, respectively; $Q(\mathbb{S})$ is the augmented metamodel. $\mathcal{V}$ is the view mapping, and $\mathcal{I}$ is an inclusion mapping denoting that $\mathbb{S} \subseteq Q(\mathbb{S})$. Note that we assume $\mathcal{V}$ to be total but not necessarily injective. This pattern is followed by all the QueST MT definitions.

### 4.2.3 Incremental Process

Given the source and the target metamodels, the user needs to complete a QueST definition by following the below process:

1. Analyze the meaning of the elements (i.e., nodes and arrows) of the metamodel $\mathbb{T}$ to find what they correspond to in model $\mathbb{S}$.

[1]Note that what we call mapping here might also be called linking in the literature.

2. For all elements in $\mathbb{T}$ with no corresponding match in $\mathbb{S}$, find a query to define it over $\mathbb{S}$; if impossible, then transformation fails.

3. Augment $\mathbb{S}$ to $Q(\mathbb{S})$.

4. Map $\mathbb{T}$ to $Q(\mathbb{S})$.

In the above process, Step 3 is a heuristic task and could be carried out incrementally as follows. $Q(\mathbb{S})$ might include the definition of multiple queries, say $Q_1$ to $Q_n$. Not all of these queries need to be defined over the original source metamodel; that is, each query definition might use nodes and arrows added to the metamodel by the previous queries. If the notation $Q_1(\mathbb{S})$ is fixed to denote the augmented metamodel after a definition of $Q_1$ over the metamodel $\mathbb{S}$, then by the definition of a series of queries $Q_i(i = 1..n)$ over $\mathbb{S}$, the final augmented metamodel will be denoted as $Q_n(..(Q_1(\mathbb{S}))..)$. Therefore, $Q(\mathbb{S})$ in Fig. 4.7 would be equal to $Q_n(..(Q_1(\mathbb{S}))..)$. Note that both the value of $n$ and the type of each individual query operation $Q_i$ vary for each particular MT definition. For example, since we used three queries in the HappyPeople example definition, we would have $Q(\mathbb{S}) = Q_3(Q_2(Q_1(\mathbb{S})))$. Note the important property of the QueST queries –i.e., their application over a metamodel augments it to another metamodel[1]– allows this chaining of query definitions consecutively.

.

## 4.3 MT Execution in QueST

The target model is generated following two consecutive phases: 1) Execution of the queries in the corresponding MT definition over the source model. 2) Retyping of the query results in the previous phase to produce the target model elements. In the following, we explain these two phases.

### 4.3.1 Query Executions

After an MT is defined at the metamodel level, any provided valid source model can be translated to the corresponding target model using the definition. Suppose that we have the definition in Fig. 4.7 and a model $S$ typed over the source metamodel $\mathbb{S}$ is provided; thus, we will have the structure shown in Fig. 4.8. $\mathcal{T}_1$ is the typing mapping specifying the types of the elements in $S$ (see Chapter 3 for more details about typing mappings). Recall that $Q(\mathbb{S})$ in Fig. 4.8 represents the definition of a series of queries; that is, $Q(\mathbb{S}) = Q_n(..(Q_1(\mathbb{S}))..)$. All of these queries are consequently executed over the provided model $S$ starting from $Q_1$. The execution of each query

[1]See Definition 12 in Sect. 3.2

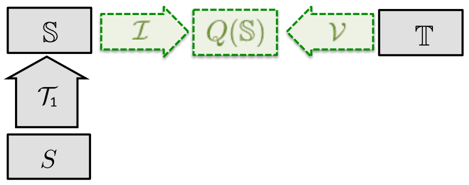


Figure 4.8: An MT execution initial structure

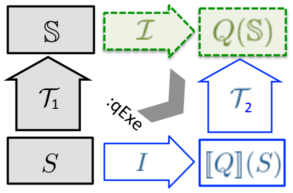


Figure 4.9: Execution of all queries

adds new elements to the model. After the execution of all queries, we will have a structure that contains all the query results as well as the source model elements. We abstractly depicted this structure by a block square labeled as $[\![Q]\!](\mathbb{S})$ in Fig. 4.9[1]. $[\![Q]\!](\mathbb{S})$ connects to the initial diagram via two mappings as seen in the figure: 1) $\mathcal{T}_2$ is a typing mapping associating the query results (and the original elements) with their corresponding types in the augmented metamodel; 2) $I$ is an inclusion mapping explicitly specifying that the main model elements are included in $[\![Q]\!](\mathbb{S})$. The entire execution process is abstractly shown by a chevron labeled as ":qExe" in the figure.

As an example, let us consider that the source model in Fig. 4.10(a) is provided as an input to the transformation definition in Fig. 4.6. Fig. 4.10(b) shows the result of the first query (i.e., $Q_1$) execution on this model. Recall that $Q_1$ is the intersection of `likes` and `owns`. Since $p1$ in the model owns and likes $c1$, it will have a `Qboth` relation with $c1$. Similarly, $p3$ will have a `Qboth` relation with $c3$ as it both owns and likes $c3$. Thus, as seen, two dashed blue arrows (typed as *Qboth*) are created as the result of $Q_1$ execution in Fig. 4.10(b).

The next query (i.e., $Q_2$) execution applies on the result of the previous query. $Q_2$ is the Domain Selectionquery and selects the `Qboth` domain and calls it `QPerson`. $p_1$ and $p_3$ comprise the `Qboth` domain; thus, as seen in Fig. 4.10(c), two new nodes typed as `QPerson` are added to the model; Accordingly, these two nodes are connected to their corresponding origins (i.e., $p1$ and $p2$) via two new arrows typed as `QisA` (see $Q_2$ definition in Page 56).

[1]In a color display or print, the reader might expect the following semantics for the colors in this figure and almost all other figures and diagrams in this thesis: black (or grey) implies elements that are given, green implies elements produced by the user, and blue implies elements automatically generated (i.e., algebraically derived).

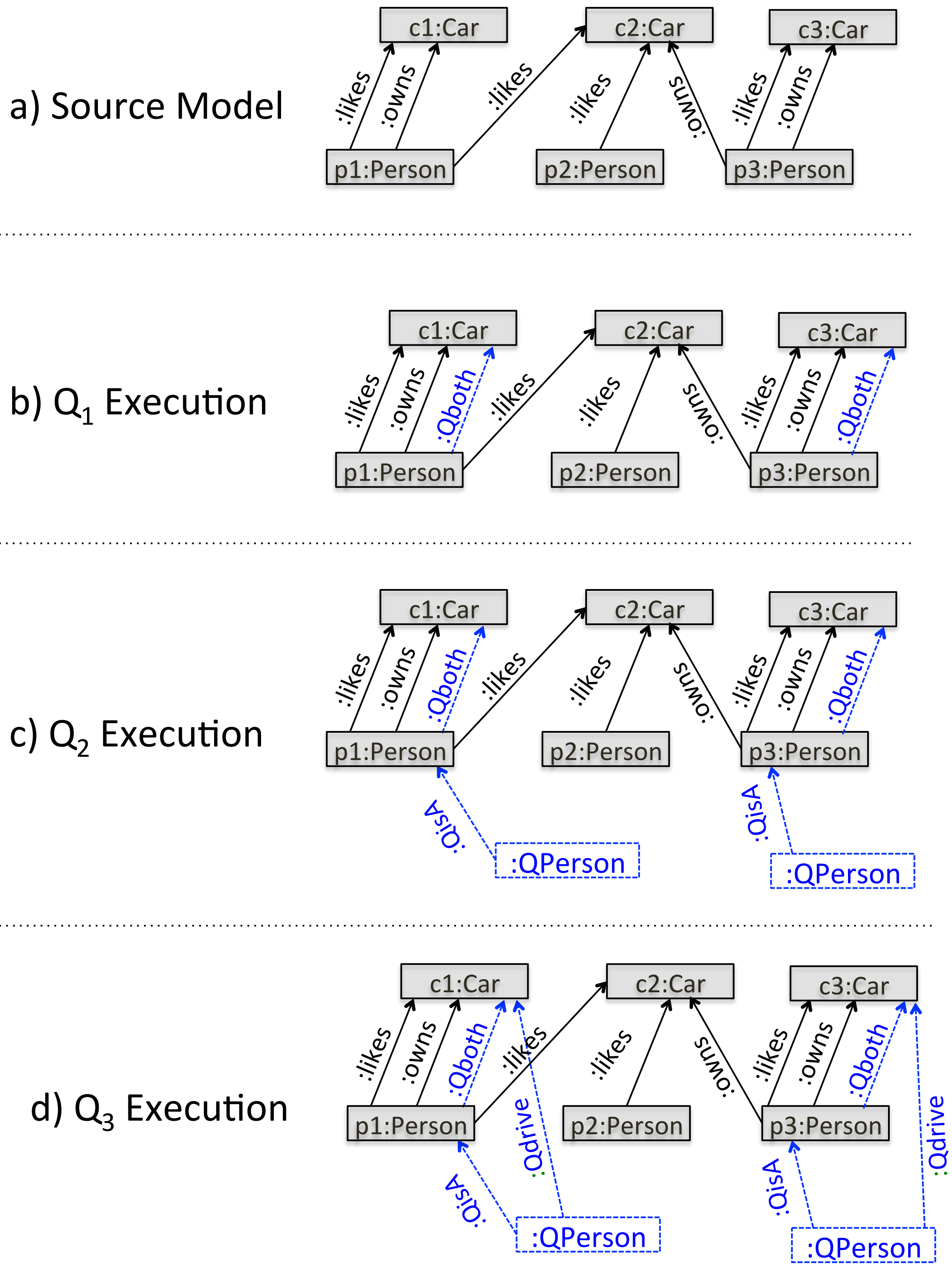


Figure 4.10: HappyPeople definition execution over a provided sample source model

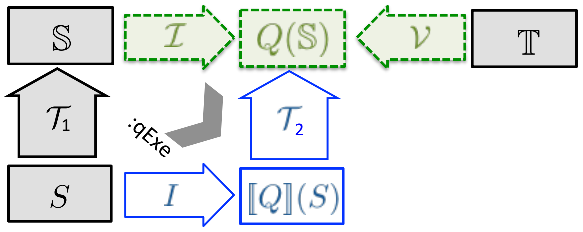


Figure 4.11: The structure before relabelling process

Finally, the last query (i.e., $Q_3$) executes. This query is the composition of `Qboth` and `QisA`. Thus, two new arrows both typed as `Qdrives` (i.e., the query result type) are added to the model as illustrated in Fig. 4.10(d).

### 4.3.2 Retyping Process

After all of the queries are executed, we will have the structure shown in Fig. 4.11. The upper part of the figure is the MT definition (see Fig. 4.7) and the left square is the result of the query execution phase (see Fig. 4.9). In the retyping phase, the target model elements are generated by relabelling the elements in the augmented model (i.e., $[\![Q]\!](\mathbb{S})$). This relabelling/retyping process is carried out according to the *view mapping* (i.e., $\mathcal{V}$) definition in Fig. 4.11.

The semantics of the relabelling process can be concisely described as a pull-back (Ehrig *et al.*, 2006) operation over the diagram $[\![Q]\!](\mathbb{S}) \rightarrow Q(\mathbb{S}) \leftarrow \mathbb{T}$ in Fig. 4.11. Thus, this figure is completed to obtain the one depicted in Fig. 4.12. The ":rType" chevron in this figure abstractly represents the retyping (i.e., pullback) operation. $T$ is the generated target model, $\mathcal{T}_3$ is its corresponding typing mapping, and $V$ is the traceability mapping tracing back the generated elements to their origins. Note that the category of graphs does have pullbacks (Ehrig *et al.*, 2006) and we can always compute ":rType". However, there might be a question whether the model produced by this pullback satisfies the constraints of the target metamodel. Here we indeed need to require good properties of $\mathcal{V}$ (i.e., compatibility of $\mathcal{V}$ with constraints in $\mathbb{T}$ and Q($\mathbb{S}$) –see Diskin and Wolter (2008)), but what we actually need is good properties of queries that would provide these good properties of $\mathcal{V}$.

The machinery of the pullback operation ensures that the generated targets are always structurally correct (i.e., they are pre-instances). However, the checking whether they are also valid models is an additional step that can be carried out as follows[1]. All the target metamodel constraints are expressed as a set of properties over the QueST MT definition. This way, they can be verified using the techniques explained

[1]In the current MT approach –such as ATL and ETL– MT definitions do not ensure the generated targets will be pre-instances. The user needs to verify this after every transformation execution.

in Chapter 6.

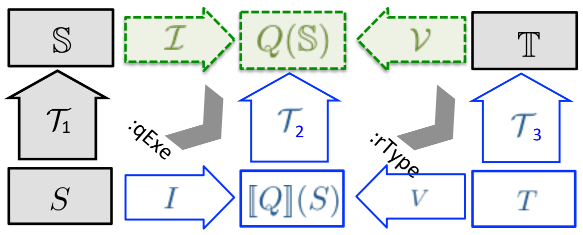


Figure 4.12: Query Execution and Retyping operations

The ":rType" operation in Fig. 4.12 simply performs the following action: the elements of $[\![Q]\!](S)$ that are typed over an element falling at the image of the view-mapping $\mathcal{V}$ are picked as elements in the target model; then, the type of these elements are changed to the corresponding types in the target metamodel. More precisely, the following pseudo code imperatively specifies the relabelling procedure.

```
1 foreach ($t$ in $\mathbb{T}$) {
2         foreach ($s$ in $[\![Q]\!](S)$) {
3                 if ($\mathcal{T}_2(s) == \mathcal{V}(t)$){
4                         add $s$ to $T$;
5                         $\mathcal{T}_3(s)$ = $t$;
6                 }
7         }
8 }
```

In the above pseudo code, lines 1 and 2 iterate over the elements in the target and the augmented source metamodel, respectively. Then, if the type of an element in the augmented metamodel is equal to an element in the $\mathcal{V}$ codomain (see line 3), that element is added to the target model (line 4), and its type is changed accordingly (line 5).

Fig. 4.13 provides a concrete illustration of the running example structure before the relabelling process. This figure shows the internal structures of the abstract node blocks in Fig. 4.11. However, except the internal links of the view mapping $\mathcal{V}$, the internal links of the other block arrows are either hidden entirely or shown using colons to avoid cluttering. For example, the typing mapping links of the source model and the augmented model are shown by colons and the mapping links between the source metamodel and the augmented metamodel are hidden. After an application of the above relabelling procedure on the structure in Fig. 4.13, the elements in the target model will be generated as seen in Fig. 4.14. As seen, five nodes and two arrows are added to the target model. Since *HappyPerson* is mapped to *QPerson* in the MT definition, all the elements in the augmented model with the type *QPerson* should

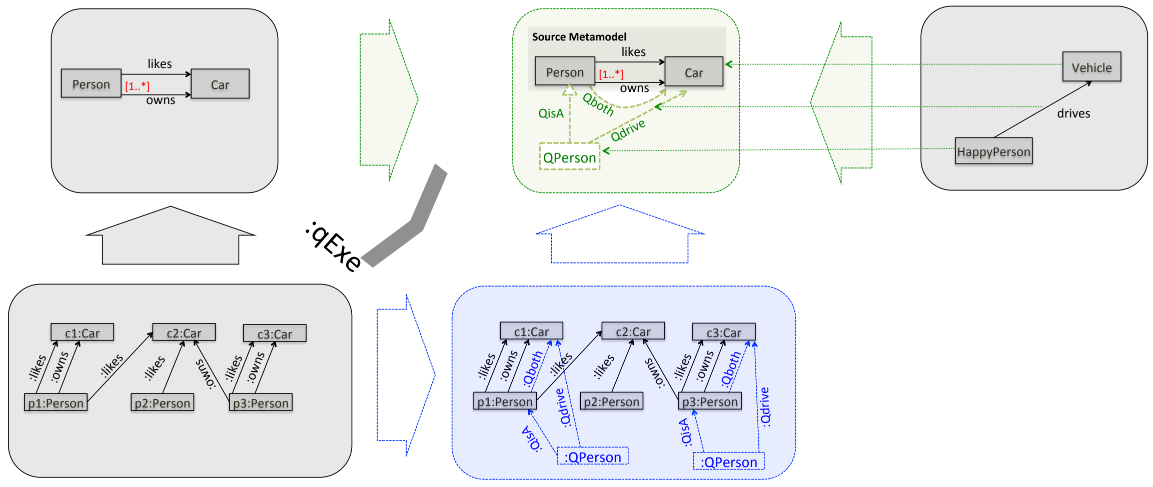


Figure 4.13: The query execution phase for the running example

be added to the target model and their type should change to $HappyPerson$; thus, two nodes $h1$ and $h2$ are added because of the two nodes of type “QPerson” in the augmented model. Similarly, the three nodes $v1$, $v2$ and $v3$ are added to the target model, because of three cars, and, finally, two arrows $d1$ and $d2$ are added because of the two arrows of type “Qdrives” in the augmented model. The entire blue block (with its respective typing mapping) constitute the generated target model for the provided source model.

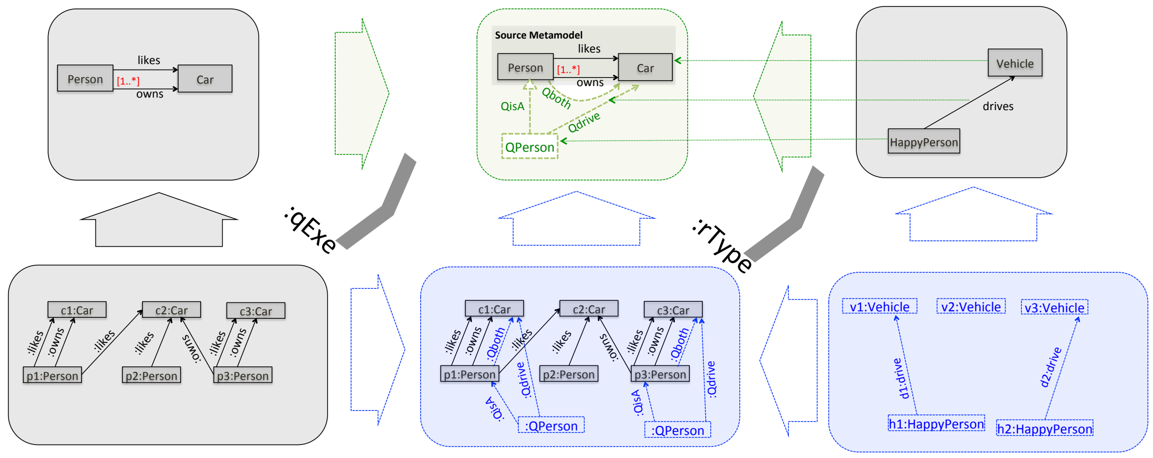


Figure 4.14: Target Model Generation.

## 4.4 Diagrammatic Queries Explained

In the previous sections, we explained that the QueST queries augment metamodels by adding new elements to them. In the following, we will explain that these queries are diagrammatic operations ($\mathscr{D}$*queries*); that is, their input and output arities are graphs that are related by an inclusion homomorphism. We will also demonstrate that the query applications are expressible by graph transformations, and their executions construct pullback completion diagrams.

### 4.4.1 Queries' Syntax

Recall from the HappyPeople example that the first query $Q_1$ takes the `likes` and the `owns` associations, and returns an association named `Qboth` (see $Q_1$ definition at page 56.) Syntactically, this intersection operation is applicable, provided that the two input associations share common source and target domains. More precisely, $Q_1$ is applicable when its input elements are in the form of the graph ${Q_1}^{in}$ illustrated in Fig. 4.15. ${Q_1}^{in}$ consists of two arrows $R_1$ and $R_2$ with their corresponding domains and codomains (labeled as $X$ and $Y$, respectively). We call ${Q_1}^{in}$ the input arity of the `arrowIntersection` operation. This operation application changes ${Q_1}^{in}$ to another graph ${Q_1}^{out}$ illustrated in Fig. 4.15; that is, it adds one arrow $R_3$ to the input arity whose domain and codomain are $X$ and $Y$, respectively. As seen in Fig. 4.15, the output arity includes the input arity element; this is explicitly expressed by the dotted links connecting the original elements in the input to their corresponding elements in the output arity graph. The entire diagram in Fig. 4.15 is called the diagrammatic arity of the `arrowIntersection` operation.

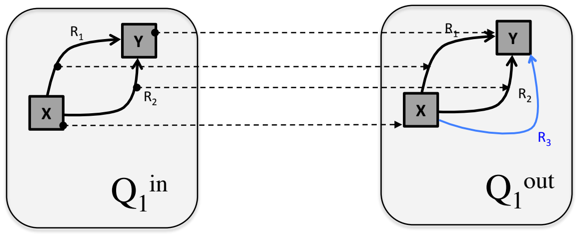


Figure 4.15: The arrowIntersection operation arity

Similar to the `arrowIntersection` operation above, the other operations used in the HappyPeople example are also diagrammatic. The diagrammatic arity of the $Q_2$ (i.e., `domainSelection`) operation is depicted in Fig. 4.16. As seen, the input arity ${Q_2}^{in}$ consists of an arrow $R_1$ with its corresponding domain $X$ and codomain $Y$. The operation takes this arrow and adds two new elements: 1) The node $Z$, and 2) The arrow $R_2$ between $Z$ and $X$.

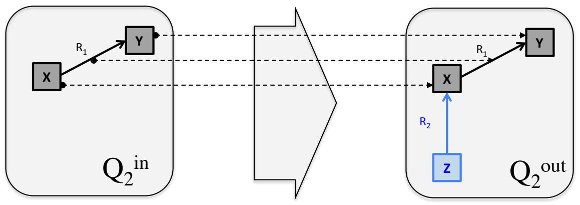


Figure 4.16: The `domainSelection` operation arity

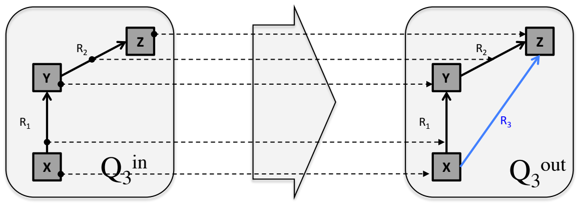


Figure 4.17: The `arrowComposition` operation arity

Finally, Fig. 4.17 depicts the diagrammatic arity of the last operation $Q_3$ (or `arrowComposition`). The operation input arity consists of two arrows $R_1$ and $R_2$ such that the codomain of $R_1$ coincides with the domain of $R_2$. The output arity adds a new arrow called $R_3$ connecting the domain of $R_1$ to the codomain of $R_2$.

### 4.4.2 Metamodel Augmentation

To define a $\mathscr{D}query$ on a metamodel, the metamodel should be marked with the $\mathscr{D}query$ input arity. Marking of a metamodel is precisely defined as a graph homomorphism whose domain is the input arity of the operation. For example, Fig. 4.18 depicts the marking of the source metamodel of the HappyPeople example with the `arrowIntersection` input arity. $X$ is mapped to $Person$, $Y$ is mapped to $Car$, and $R_1$ and $R_2$ are mapped to the *like* and the *owns* associations, respectively. Note that the graph homomorphism indicating the markings does not need to be an injection. For example, Fig. 4.19 illustrates a marking that is not injective, since $Person$ is marked as both $X$ and $Y$. The associations *married* and *loves* are marked as $R_1$ and $R_2$ is this figure, respectively.

Upon defining a marking $M : Q^{in} \to G_{\mathbb{M}}$ over a metamodel graph $G_{\mathbb{M}}$ using a query $Q : Q^{in} \to Q^{out}$, the metamodel is augmented and new elements are defined automatically. This augmentation process is precisely defined as a graph transformation with the match $Q^{in} \to G_{\mathbb{M}}$ and the rule $Q^{in} \to Q^{out}$; that is, the arity of the $Q$ operation acts as the rule of the transformation operation and transforms the graph $G_{\mathbb{M}}$ based on the match $Q^{in} \to G_{\mathbb{M}}$. The transformation operation is a single pushout operation (in the category of graphs and homomorphisms) and can be graphically

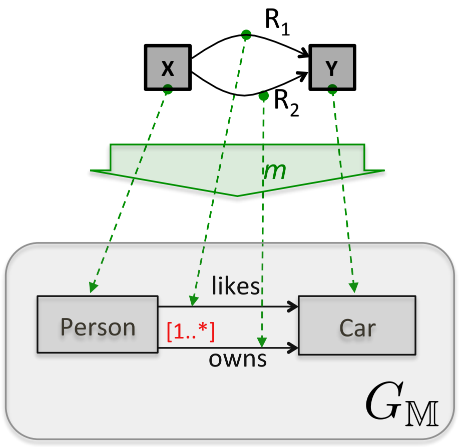


Figure 4.18: Explicit marking of a metamodel

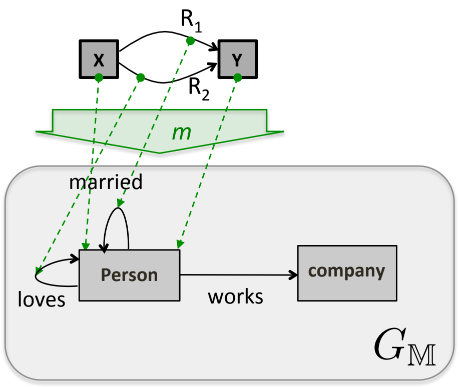


Figure 4.19: Non-injective marking of a metamodel

depicted as follows:

$$\begin{tikzcd}
Q^{in} \arrow[r, hook, "i"] \arrow[d, "m"'] & Q^{out} \arrow[d, "m'"] \\
G_{\mathbb{M}} \arrow[r, hook, "i'"'] & G'_{\mathbb{M}}
\end{tikzcd}$$

In the above diagram, the nodes are graphs and the arrows are graph homomorphisms. Since $Q^{in} \subseteq Q^{out}$, the above pushout operation always augments the metamodel by adding new elements (or leaves it untouched). The above transformation can be imperatively specified by the following pseudo code.

```
1  //Input: $m : {Q_1}^{in} \to G_{\mathbb{M}}, i : {Q_1}^{in} \to {Q_1}^{out}$
2  //Ouput: $m' : {Q_1}^{out} \to G'_{\mathbb{M}}$, $i' : G_{\mathbb{M}} \to G'_{\mathbb{M}}$
3  //-------------------------
4  var $i'$=new identity($G_{\mathbb{M}}$)   //create an identity arrow
5  var $G'_{\mathbb{M}}$= $i'.codomain$            //create a copy of $G_{\mathbb{M}}$
6  var $m'$ =new homorphism($m$)         // create a copy of m
7  moveDomainAndCodomain($m, i, i'$)
8
9  var newNodes=${Q_1}^{out}.Nodes - {Q_1}^{in}.Nodes$
10 var newArrows=${Q_1}^{out}.Arrows - {Q_1}^{in}.Arrows$
11 foreach (node in newNodes) {
12         var n=new Node()   // create a new node
13         n.label=m.label+"."+node.label  //set the label
14         $G'_{\mathbb{M}}.Nodes.add(n)$
15         $m'.add(node \to n)$
16 }
17 foreach (arr in newArrows) {
18         var a=new Arrow()       // create a new arrow
19         a.label=m.label+"_"+arr.label  //set the label
20         $a.src = m'(arr.src)$
21         $a.trg = m'(arr.trg)$
22         $G'_{\mathbb{M}}.Arrows.add(a)$
23         $m'.add(arr \to a)$
24 }
25 return $m' : {Q_1}^{out} \to G'_{\mathbb{M}}$, $i' : G_{\mathbb{M}} \to G'_{\mathbb{M}}$
```

In the above pseudo code, we first create an identity arrow over $G_{\mathbb{M}}$ and call it $i'$ (line 4). Then, we assign its codomain to $G'_{\mathbb{M}}$ (line 5). Then, we create a copy of homomorphism $m$ and call it $m'$. Line 7 moves the domain and codomain of $m'$ along two injective arrows $i$, and $i'$, respectively. Then, we calculate the difference between ${Q_1}^{out}$ and ${Q_1}^{in}$ (lines 9–10). The two *for* loops (lines 11–23) iterate over the nodes and arrows in the difference list, and add elements to $G'_{\mathbb{M}}$ and set the corresponding mappings from $m'$ to those elements, accordingly. Note that the arrow $i'$ does not change during the process. The return value of the pseudo code (line 25) includes the elements shown in blue in the pushout diagram presented earlier.

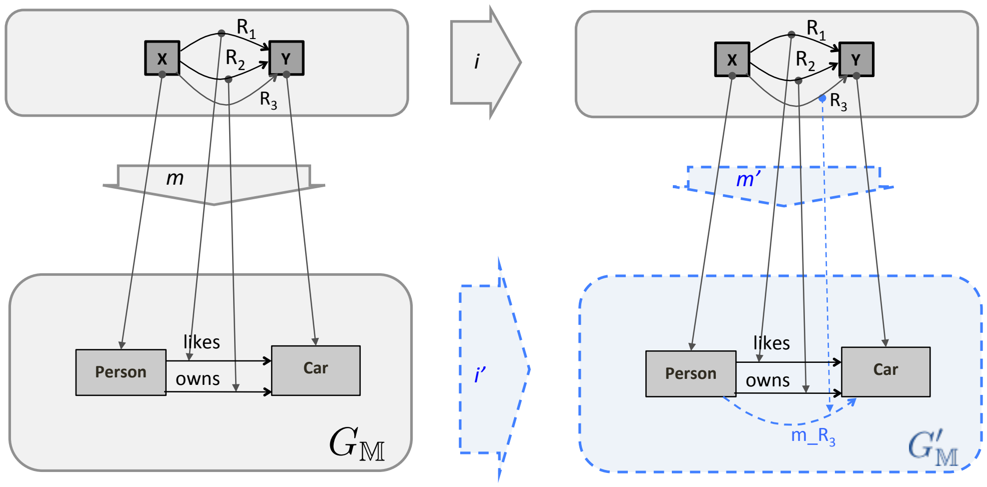


Figure 4.20: Augmentation process generates an arrow

The grey block elements in Fig. 4.20 illustrate the initial step before applying the above pseudo code to the marking of the HappyPeople example source metamodel with the $Q_1$ operation. The blue dotted block elements in the figure are those that are created after the execution of the pseudo code. For this specific example, the difference between $Q^{out}$ and $Q^{in}$ is just an arrow (i.e., $R_3$); thus, a new arrow (i.e., $m\_R_3$) is added to the metamodel $G'_{\mathbb{M}}$ (see line 20 in the pseudo code), and an appropriate mapping link connecting $R_3$ to $m\_R_3$ is generated (see line 21). Note that users can change the generated element names; for example, we renamed $m\_R_3$ to *Qboth* when we were initially explaining the $Q_1$ application at the HappyPeople example (see Fig. 4.6).

### 4.4.3 Sequence of Query Definitions

Recall from Sect. 4.2.3 that query operations can be applied in a sequence to incrementally augment a metamodel; that is, the entire or parts of the outputs of one operation can be used as inputs of the next. We have already seen the explicit marking of a metamodel with the `arrowIntersection` operation in Fig. 4.18 and its respective augmentations in Fig. 4.20. $M\_R_3$ in this figure is the output of the query application. In the following, we demonstrate how this arrow is used as an input to the next operation and, consequently, the output of the second operation is used as an input of the third operation. The forthcoming figures will show the explicit markings and the augmentation processes in this series of query applications (note that $M\_R_3$ is called *Qboth* in these figures).

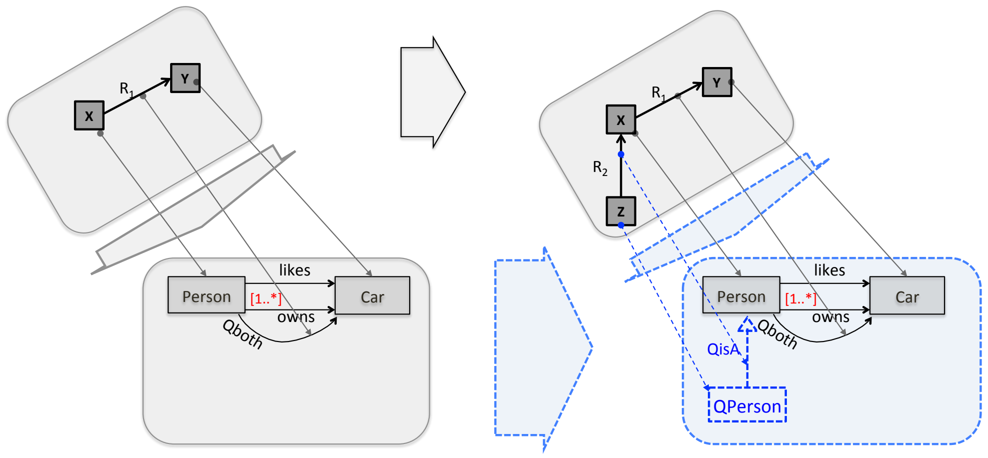


Figure 4.21: Marking of the metamodel with Domain Selection query and its augmentation

Fig. 4.21 shows that the new element *Qboth* is used as the `domainSelection` operation input: $X$ and $Y$ are matched with $Person$ and $Car$, respectively; and $R_1$ is matched with the *Qboth* association. After this matching, the graph transformation is executed and the node $QPerson$ and the arrow $isA$ are added to the metamodel (see the right-hand side of the figure). Consequently, Fig. 4.22 shows the way the input arity of the `arrowComposition` operation is matched with the elements of the augmented metamodel that are generated in the previous steps: $X$, $Y$, and $Z$ are matched with, respectively, $QPerson$, $Person$, and $Car$; and $R_1$ and $R_2$ are matched with the two generated elements from the previous steps, namely, $QisA$ and *Qboth*. The application of this query generates a new arrow called $Qdrives$ between $QPerson$ and $Car$. This operation completes the augmentation process in the HappyPeople example and the elements in the target metamodel are mapped to the elements in this augmented metamodel (see Fig. 4.6 for this view-mapping).

## 4.4.4 Queries' Semantics

The syntax of a query specifies the shapes of its input variables and also the shape of its generated values. The semantics of a $\mathscr{Q}query$ defines the values for the outputs, given that the input variables are bound to some values.

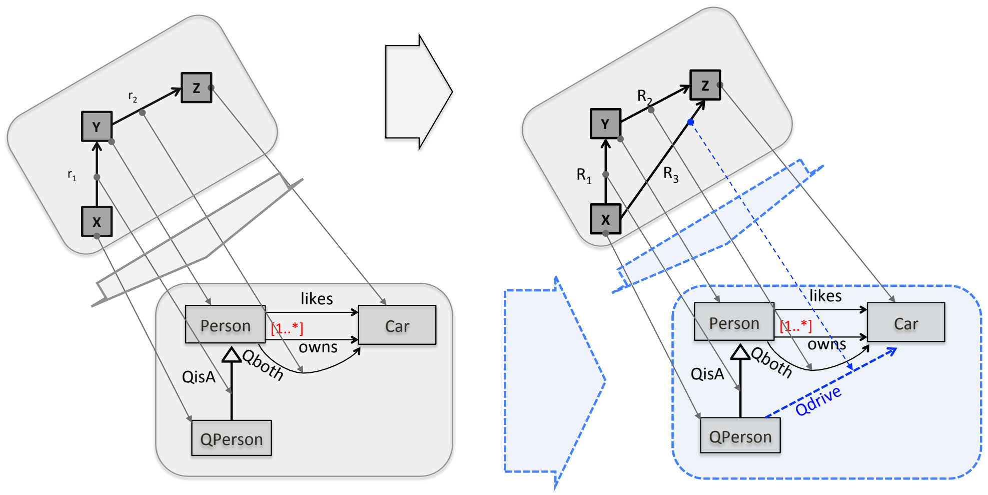


Figure 4.22: Marking of the metamodel with Arrow Composition query and its augmentation

When a query is defined on a metamodel, all elements/variables of its input arity are matched with some elements in the metamodel. For example, as seen in Fig. 4.23, the input variables of the `arrowIntersection` operation are matched as follows: $X$, $Y$, $R_1$ and $R_2$ are matched with $Person$, $Car$, $likes$, and $owns$, respectively. Now suppose a model typed over this metamodel is provided as seen in the same figure. In this model, John and Sara are typed over Person. The collection consisting of these two people/nodes is a value for the variable $X$, since $X$ is matched with $Person$. In other words, we can say that the variable $X$ is bound to the collection $\{John, Sara\}$. Similarly, the arrows in the input arity should also be bound to some values. For example, since $R_1$ is matched with $likes$, and the references $l_1$ and $l_2$ are both typed over $likes$, the collection consisting of the latter two references is the value of the variable $R_1$. In other words, we can say the arrow $R_1$ is bound to the collection $\{l_1, l_2\}$ during the query execution. Following this approach, the complete list of bindings for the provided match and the model in Fig. 4.23 will be as follows:

- $X = \{John, Sara\}$
- $Y = \{BMW, Ford\}$
- $R_1 = \{l_1, l_2\}$
- $R_2 = \{o_1, o_2\}$

Following the above approach, we can always get to the values of the input variables of a query $Q$. These values can be concisely specified as a homomorphism shown

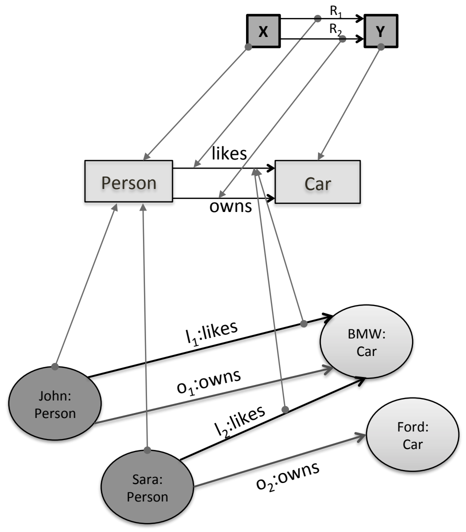


Figure 4.23: The variable binding in the arrowIntersection $\mathscr{D}query$

in the following[1]:

$$\begin{tikzcd} Q^{in} \\ O_* \arrow[u, "b_{in}"] \end{tikzcd}$$

Assuming the above value for the $Q$ input variable, the $Q$ semantic is defined as a function $f : \mathbb{Q}^{in} \rightarrow \mathbb{Q}^{out}$ that reads the value $b_{in} \in \mathbb{Q}^{in}$, and generates an output $b_{out} \in \mathbb{Q}^{out}$ (that is also a homomorphism) such that $b_{in} \subseteq b_{out}$ (see Sect. 3.2.2 for a more precise definition). For example, the `arrowIntersection` semantics can be defined as the following:

```
//Input: ($R_1 : X \rightarrow Y$, $R_2 : X \rightarrow Y$)
//Ouput: ($R_1 : X \rightarrow Y$, $R_2 : X \rightarrow Y$, $R_3 : X \rightarrow Y$)
R_{3} = new Arrow(X,Y)  // create an empty Arrow
foreach ($r_1$ in $R_1$) {
  foreach ($r_2$ in $R_2$) {
    if ($r_1$.src=$r_2$.src $\wedge$ $r_1$.trg=$r_2$.trg){
      // create a new reference of type $R_3$
      var $r_3 = new\ reference(R_3)$
      $r_3$.src=$r_1$.src
```

[1]This value homomorphism is produced by the pullback operation as will be explained in Sect. 3.2 in a precise way.

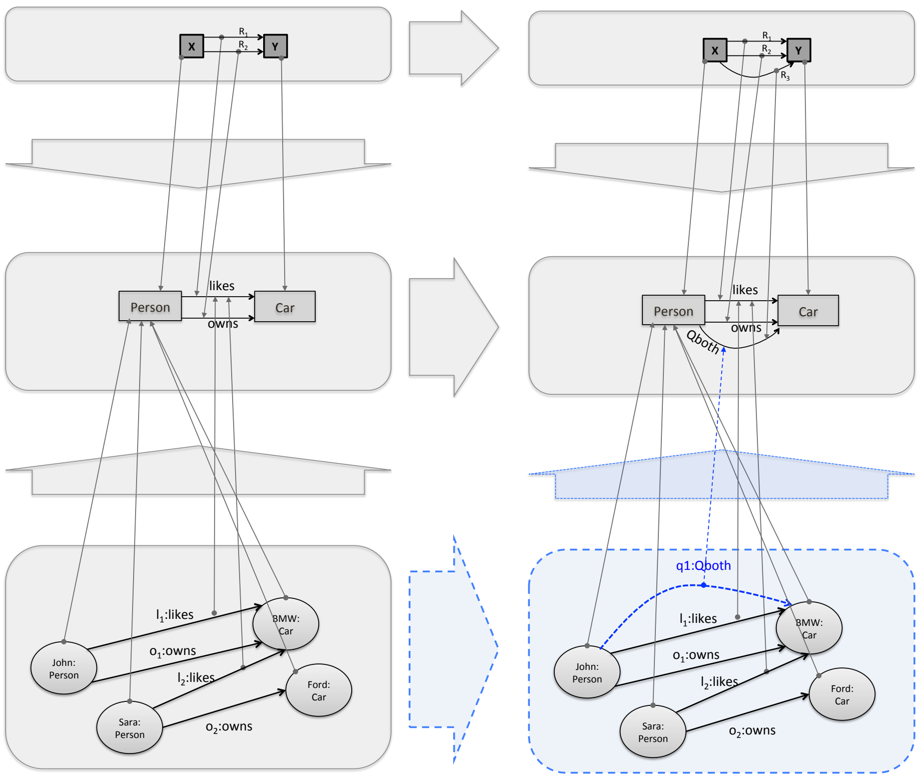


Figure 4.24: An example of a query execution over a metamodel

```
10          $r_3$.trg=$r_1$.trg
11          $R_3.add(r_3)$
12        }
13     }
14  }
15  return $R_3$
```

The above pseudo code loops over the $R_1$ collection. For each $r_1$ reference in this collection between the two nodes $x_1$ and $y_1$, whenever it finds a similar reference in $R_2$ (that is, a reference connecting the exact same nodes), it creates a new reference $r_3$ of type $R_3$ linking the same exact nodes again (see lines 7 to 10) and adds it to $R_3$ at line 11. It finally returns $R_3$ (line 15). If we execute the above pseudo code for the configuration in Fig. 4.23, a new reference connecting *John* to *BMW* will be generated as seen in Fig. 4.24. This new reference typed over *Qboth* is distinguished with a blue dashed arrow in the figure.

$\mathscr{D}$*query* definitions are MT independent; that is, they are defined once in the DQF framework (see Definitions 9 and 14) and used over and over again in QueST MT definitions. Thus, it is important to distinguish the following two actions: 1) the definition of a DQF query that happens at the DQF framework level and is independent of any specific MT definition, 2) an application of such a DQF query to an MT definition (Definition 12). Thus, the user defines the syntax and semantics of a query once and will use it multiple times or share it with others. For example, the syntax and semantics of the `arrowIntersection` query are defined once independently of any MT definition; however, the query can be used multiple times in any MT definition.

## 4.5 Conclusion

In this chapter, we presented the way an MT can be defined and executed in the QueST approach. We provided an abstract picture of a QueST MT definition and execution, and explained the concepts concretely using a simple example. The main component of a QueST definition is a series of queries that are defined on a source metamodel and extending it with adding new elements. This procedure is guided by the structure of the target metamodel with the goal of replicating this structure on the source side. After necessary queries are defined, the elements in the target metamodel are mapped to the query results or the original source elements. The entire MT definition process occurs on the metamodel layer. Thus, after an MT is defined, it can be executed over any model conforming to the source metamodel in the definition. To execute the definition, first, the defined queries are executed over the source model; then, the target model elements are created in a straightforward relabelling procedure using the final result of the query execution procedure in the context of the MT definition structure.

The queries that are used in building transformations in QueST are not similar to ordinary query operations: 1) they are high-level operations applied on the metamodel elements; 2) they have diagrammatic arities. In Chapter 3 we defined the DQF framework at which the syntax and semantics of these queries are specified. In this chapter we illustrated their application in the definition of a QueST MT. The query operations in DQF can be defined independently of any MT definition employed in each concrete QueST MT definition. This enables the creation of a library of diagrammatic queries that can be shared among the MT community. As the DQF queries have homogenous input and output arities (they are both graphs), they can be applied in a chain and make it possible to define MTs incrementally. The query chaining mechanism all happens in the metamodel layer, without interfering in their execution on any particular model.

# Chapter 5

# QueST vs. Rule-Based MT Approaches

"Simplicity is prerequisite for reliability."

Edsger Dijkstra

There exist a variety of model transformation languages as we mentioned earlier in Chapter 1. We placed them within the two categories of rule-based and graph-based MT languages and briefly discussed their properties in Sect. 1.3. In this chapter, we study the transformations written in QueST and contrast them with the ones written in rule-based MT approaches ETL, ATL, and QVT-R. The dimensions along which we want to compare these approaches with QueST are declarativity, incrementality and modularity. More specifically, the questions we want to answer are as follows: 1) Is QueST more declarative than these rule-based approaches? 2) Can we build MT definitions by incrementally composing main structural components in each of these approaches? 3) Are semantically related queries also syntactically collocated in the MT definitions? Recall from Chapter 1 that MT is one of the key operations involved in maintaining the network of inter-related models. We believe that all the above aspects (i.e., declarativity, incrementality, and modularity) for an MT are essential in effective maintenance of such a network. We will answer the above questions by studying the main structural components of these approaches and their corresponding input and output parameters. To put our study in a context we will implement two examples in all of the four approaches (i.e., QueST, ETL, ATL, and QVT-R) and will refer to these implementations in our discussions.

In the first part of this chapter, we provide implementations of the examples. We will briefly explain the main structural components of ETL, ATL, and QVT-R and provide the implementation of the HappyPeople example in these languages. (The

QueST implementation is already provided in Sect. 4.1). Then, we introduce a richer example called ClassToTable and implement it in QueST and each of the three rule-based approaches. We will explain the way this transformation is decomposed to rules/queries and are defined in these languages. In the second part of this chapter we conduct our analysis. We first look at the main structural component of these approaches and argue that QueST provides more declarative constructs than these rule-base approaches. Then, we will examine the relations between the structural components and source and target metamodels; such analysis will help to further understand the organization of the structural components in MT definitions. Later on, we examine wether we can compose/decompose the structural component to build MT definitions incrementally. Then, we zoom in to the contents of the structural components (i.e., rules and queries) and analyze the way QueST query contents are represented in the other implementations. Finally we will conclude the chapter and summarize our findings.

## 5.1 The HappyPeople Example

In Chapter 4, we introduced the HappyPeople example and provided its implementation in QueST. In the following sections, we provide its implementations in ETL, ATL, and QVT-R.

### 5.1.1 The HappyPeople Example in ETL

An MT in ETL is specified via a set of rules. Each rule is associated with one source type (i.e., source metamodel node) and one or more target types (i.e., target metamodel nodes) as its parameters. A rule might have a guard expression that constrains its application to a subset of source type instances. Semantically, during a transformation execution, each node in the source instance is checked against all the rules; if it matched a source type of a rule and satisfied its guard condition, one target instance is created for each of the rule's target type parameters; then the property values of the created instances are populated according to the rule body expressions. The arrows are treated as secondary citizens in ETL since their generation is expressed within the rule bodies that are written for the nodes[1].

Fig. 5.1 presents an implementation of the HappyPeople example in ETL. The implementation consists of two rules *PersonToHappyPerson* (line 1) and *CarToVehicle* (line 27). *PersonToHappyPerson* has a guard condition/constraint (lines 5–14); the constraint states that for a person to be matched by the rule, the intersection of the

[1]As we will see later, arrows are treated in this way in ATL, and QVT-R too.

```
rule PersonToHappyPerson
  transform person : SourceMM!Person
  to happyPerson : TargetMM1!HappyPerson{

  guard :
  not
    person.likes->asSet()->includingAll(person.owns->asSet())->
    excludingAll(
        person.owns->asSet()->excludingAll(person.likes->asSet())
    )->
    excludingAll(
        person.likes->asSet()->excludingAll(person.owns->asSet())
    )->
    isEmpty()

    for (car1 in person.likes){
        for (car2 in person.owns){
            if (car1==car2){
                for (v in car1.equivalents("CarToVehicle")->asSet()){
                    happyPerson.drives.add(v);
                }
            }
        }
    }
}

rule CarToVehicle
  transform car : SourceMM!Car
  to vehicle : TargetMM1!Vehicle {
}
```

Figure 5.1: An Implementation of the HappyPerson Example in ETL.

person's liked cars with the person's owned cars should not be empty. Since the intersection operation is not found directly supported in the ETL (Kolovos *et al.*, 2010), we have used a workaround (i.e., $A \cap B = ((A \cup B) - (A - B)) - (B - A))$ to define it in terms of union (see `includingAll()` at line 7) and subtraction (see `excludingAll()` at lines 8–12) operations. The rule creates a `happyperson` node for each `person` node that satisfies this expression. The body of the rule (lines 16–24) is an imperative implementation that builds the arrows of type `drives` for the generated happyperson nodes. The code is self-descriptive; the only part that needs further explanation is line 19. The `car1.equivalents("CarToVehicle")` expression returns the vehicles corresponding to `car1` that are generated by the `CarToVehicle` rule.

The second rule (i.e., `CarToVehicle`) is very simple in comparison to the first one described above. It neither has any guard expression nor any rule body. It matches all the car instances in the source and generates one corresponding vehicle for each of them.

### 5.1.2 The HappyPeople Example in ATL

ATL supports both declarative and imperative constructs in the definition of transformation rules; however, as we are interested in the most high-level structures of ATL in the transformation definitions, we will use the declarative constructs of the language to implement the HappyPeople example. A declarative MT definition in ATL consists of a set of *matched rules*. Each matched rule in ATL is interpreted as follows: if the rule source pattern is matched to some source model elements, the specified target elements will be generated, and the values specified in the rule body will be assigned to the generated elements' properties (if there were any). The rule execution is nondeterministic, so no order is assumed in matching the rules over the source model elements.

Fig. 5.2 presents an implementation of the HappyPeople example in ATL. This implementation consists of two matched rules: 1) `PersonToHappyPerson` and 2) `CarToVehicle`. Similar to ETL, the first rule translates the Person instances to the HappyPerson instances in the target. The second rule translates the Car instances to the corresponding Vehicle instances in the target. `person` in the first rule is an instance variable and refers to an instance of a Person. `SourceMM` and `TargetMM1` are references to the source and the target metamodels, respectively. Similar to ETL guard expressions, the expression enclosed in parenthesis after "SourceMM!Person" (line 6-8) is a boolean expression. The expression is written in the OCL (Warmer and Kleppe, 2000) language and limits the matched elements to those satisfying this condition. The expression is literally translated to the following semantics:

*"Interpret the collection of a person's liked cars as a set, and intersect it with the collection of the cars the same person owns interpreted as a set; if this intersection is*

```
module MT1;
create OUT : TargetMM1 from IN : SourceMM;
rule PersonToHappyPerson {
    from
        person : SourceMM!Person (
            not person.likes->asSet()->
                intersection(person.owns->asSet())
                    ->isEmpty()
        )
    to
        happyPerson : TargetMM1!HappyPerson
        (
            drives <-(person.likes->asSet()->
                        intersection(person.owns->asSet()))
        )
}

rule CarToVehicle {
    from
        car : SourceMM!Car
    to
        vehicle : TargetMM1!Vehicle
}
```

Figure 5.2: An Implementation of the HappyPerson Example in ATL.

*not empty, this person is selected."*

In ATL (similar to ETL), there are no independent rules for generating arrows. Thus, their generations are necessarily incorporated into the node generation rules. For example, as seen at lines 13–14 in Fig. 5.2, the *drive* arrow instances are generated within the `PersonToHappyPerson` rule. These lines are translated literally as follows:

*"take the cars collection that a person likes as a set, and intersect it with the cars collection the same person owns as a set; then, convert the intersection result of cars to the corresponding vehicles (that is achieved by the entire translation) and assign it to be the person's corresponding happypeson vehicles that he/she drives."*

The second rule CarToVehicle in the figure (line 19) neither has a guard expression nor contributes to the generation of any arrow instance. It simply translates all the car instances to their corresponding vehicle instances.

### 5.1.3 The HappyPeople Example in QVT-R

QVT-R (OMG, 2015) is the OMG proposed standard for defining model transformations. A QVT-R MT is defined using a set of constructs called relations in the QVT-R terminology. Each relation definition specifies how some elements in the source model are related to some elements in the target model. The relations are of two types: *top* and *non-top*. At the time the QVT-R engine executes an MT definition, it ensures that all the defined top relations hold true, by creating missing elements in the target. The enforcement of the non-top relations is triggered via top relations. Each relation might come with a *when* clause and a *where* clause which act as pre- and post-conditions, respectively, for the relation execution (Macedo and Cunha, 2013). Each relation has at least two domain blocks that specify the source and target of the relation. The contents of these domain blocks specify the pattern that is matched against the provided models.

An implementation of the HappyPeople example in QVT-R is provided in Fig. 5.3. Similar to the two previous implementations, the definition consists of two top relations `PersonToHappyPerson` and `CarToVehicle`. The first one generates instances of HappyPerson and the second one generates instances of Vehicles. The generation of the `drives` association instances is defined within the body of the first relation (see line 8). The body of the domain blocks are patterns that are matched against the provided models. For example, lines 3–6 specify a pattern that matches any person who likes and owns the same car. This is achieved by binding the same variable called `car` to both `likes` and `owns` associations (see OMG (2015) for detailed semantics of the patterns and the variable bindings). For the matched person instance, the transformation creates a corresponding `happyPerson` in a way that its `drives` association is bound to the `vehicle` variable (lines 7–9). Note that the latter two variables (i.e., `car` and `vehicle`) are used as the input parameters in the `when` clause at line 11.

```
 1 transformation HappyPerson(sourceModel:sourcemm, targetModel:targetmm1) {
 2     top relation PersonToHappyPerson {
 3       checkonly domain sourceModel person : sourcemm::Person {
 4             likes=car:sourcemm::Car{},
 5             owns=car:sourcemm::Car{}
 6       };
 7       enforce domain targetModel happyPerson : targetmm1::HappyPerson {
 8           drives=vehicle:targetmm1::Vehicle{}
 9       };
10       when{
11          CarToVehicle(car,vehicle);
12       }
13     }
14
15     top relation CarToVehicle {
16       checkonly domain sourceModel car : sourcemm::Car {
17       };
18       enforce domain targetModel vehicle : targetmm1::Vehicle {
19       };
20     }
21 }
```

Figure 5.3: An Implementation of the HappyPerson Example in QVT-R.

This means that the relation `CarToVehicle` should already be enforced (as it is the precondition of `PersonToHappyPerson`) and the values of both variables are determined by the `CarToVehicle` relation. The relation `CarToVehicle` is a simple relation and generates a vehicle for each car in the source. Note that it does not have any pre-condition (i.e., *when* clause) and the source domain (line 16-17) pattern is empty, so all the cars in the source model are matched when executing this relation.

## 5.2 The ClassToTable Example

In this section, we introduce a new example called ClassToTable that is more complex than the HappyPeople example. We will examine this example and its definitions in the rule-based and the QueST approaches, as we did for the HappyPeople example.

Class diagrams are used to specify the structure of the objects and their interrelations in software systems. To save the status of these objects in a relational database, they should be converted to records (i.e., elements of tables). More precisely, a class diagram structure should be appropriately translated to a corresponding relational schema. In the following, we first provide the metamodels of the class diagrams and the database schemas; then, we define the transformation rules. Finally, we explain the transformation implementations in QueST, ETL, ATL, and QVT-R.

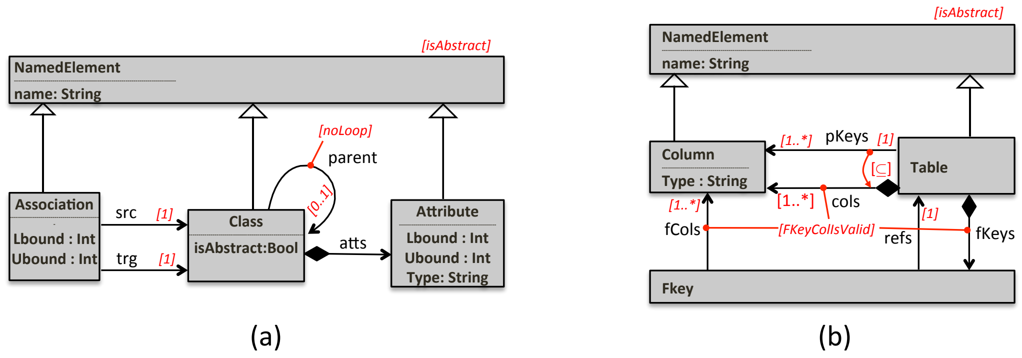


Figure 5.4: Class Diagram and Database Schema metamodels

### 5.2.1 Source and Target Metamodels

Fig. 5.4(a) exhibits the class diagram (CD) metamodel. The `Class` type represents the classes; the `Association` type represents the associations, and the `Attribute` type represents the attributes in a class diagram. All these types inherit the `name` property from the abstract type `NamedElement`.

`Class` has an `abstract` property indicating whether or not a class is abstract. The association `atts` between `Class` and `Attribute` indicates which attributes belong to a class. A class might have an inheritance relation with at most one other class; this is specified by a loop association called `parent` over the `Class` type[1]. Each attribute has a type (indicated by `Type`) and multiplicity-related properties (indicated by `Lbound` and `Ubound`). Similar to an attribute, an association has multiplicity-related properties `Lbound` and `Ubound`. Further, it should have one associated source and one associated target class; this is denoted by two associations `src` and `trg` between `Association` and `Class` in the metamodel.

The constraints associated with the metamodels are exhibited using the red enclosing brackets. Many of them are multiplicity constraints on the associations; for example, the multiplicity [1] at the end of the `src` association denotes that every association in a class diagram should have one and only one class as its source. The [0..1] constraint indicates that having a parent is not mandatory for each class, and the [1] constraint on the tail of `atts` indicates that each attribute belongs to one and only one class in a class diagram. Finally, the [noLoop] multiplicity indicates that there exists no loop in the inheritance hierarchy of the corresponding class diagram.

Fig. 5.4(b) shows the metamodel of database schemas. `Table`, `Column`, and `Fkey` represent, respectively, the tables, the columns, and the foreign keys, in a database schema (DBS). All of these types inherit the abstract type `NamedElement` that has a

[1]In UML, a class might inherit more than one class; we assumed only the possibility of a single inheritance in our simplified version.

`name` property. The `cols` association between `Table` and `Column` indicates that each table has at least one column associated with it; some of these columns are primary keys; this is indicated by another association `pKeys` between `Table` and `Column`. Each table might have foreign keys; the `fKeys` association between `Table` and `Fkey` indicates this relation. The `type` property of `Column` specifies the column type. A collection of columns might be associated with each foreign key; this is indicated by the `fCols` association between `Fkey` and `Column`. Finally, the `refs` association between the `Fkey` and the `Table` type indicates to which table each foreign key refers.

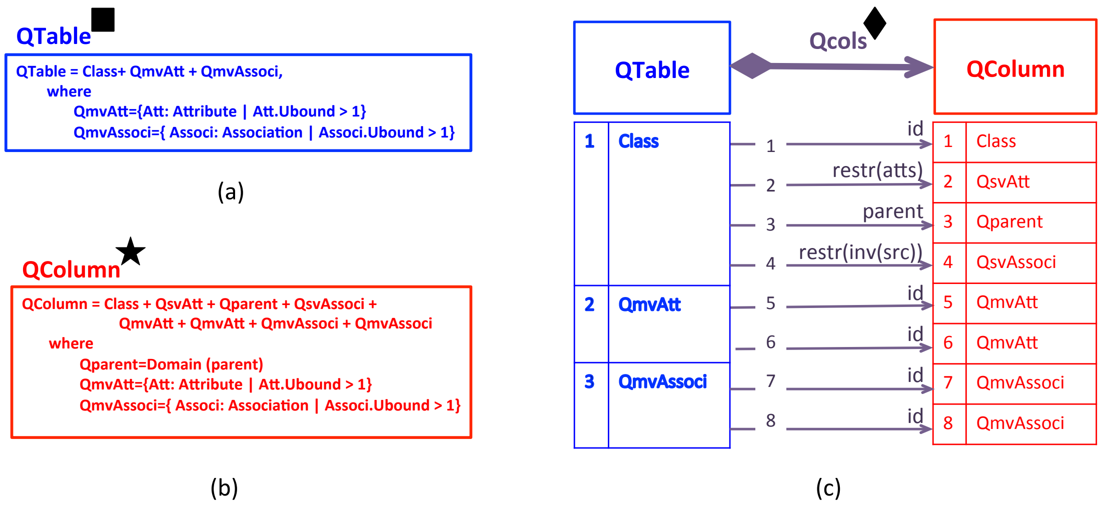
The constraints are indicated by red square-bracketed texts connected to the metamodel elements. The [$\subseteq$] constraint specifies that the primary keys are columns of the same tables. The `[FKeyColIsValid]` constraint states that the columns to which each foreign key refers are a subset of table columns of which they are a part (i.e., `fKeys`;`fCols` $\subset$ `cols`). The remaining constraints on the metamodel are the multiplicity constraints; for example, the [1] constraint on the `refs` association indicates that every foreign key should refer to one and only one table. The other multiplicity constraints are similarly interpreted.

### 5.2.2 ClassToTable Transformation Rules

There are different ways to translate a class diagram to its corresponding database schema (Embley and Thalheim, 2012). We propose the following rules as the transformation specification. For each class, we generate a table. Single-valued attributes —those with a multiplicity of one or zero— of a class are translated to columns of the corresponding table. A table is generated for each multi-valued attribute —those with the multiplicity greater than one. These tables have two columns: one for keeping the attribute values and another used as a foreign key referring to the table corresponding to the attribute containment class. Single-valued associations (svAssoci) are handled by foreign keys; for each svAssoci, we create a column in the table corresponding to the source of the association. For each multi-valued association (mvAsspci), we create a table with two foreign keys which refer to the source and the target of the association, respectively. Inheritance is handled in a way similar to the single-valued associations.

### 5.2.3 ClassToTable in QueST

As we already explained in Chapter 4, queries are the main building blocks of an MT definition in QueST. These queries are built based on the following thought process: *Which elements in the source model generate elements of a specific target type?* This arises from the fact that each target type (i.e., target metamodel element) is mapped to a query result defined on the source metamodel. This query collects the information that is required to build the target elements; for example, according to the MT rules specified in Sec. 5.2.2, tables in a database schema are generated in three different ways: 1) for each class, 2) for each multi-valued attribute, and 3) for each multi-valued association. This can be specified by defining a query on the source metamodel. Fig.5.5(a) exhibits this query definition in mathematical notation. `Class` is the collection of all classes in the model, `QmvAtt` is the collection of all multi-valued attributes and `QmvAssoci` is the collection of all the multivalued associations. The plus (+) notation denotes the disjoint union operation and the expressions in the "where" clause select the `QmvAtt` and the `QmvAssoci` collections. The ■ , ♦ and ★ signs on the `Qtable`, `Qcol` and `QColumn` in this figure will be used for comparison purposes in Section 5.3.4.


QTable■

QTable = Class+ QmvAtt + QmvAssoci,
where
QmvAtt={Att: Attribute | Att.Ubound > 1}
QmvAssoci={ Associ: Association | Associ.Ubound > 1}

(a)

QColumn★

QColumn = Class + QsvAtt + Qparent + QsvAssoci +
QmvAtt + QmvAtt + QmvAssoci + QmvAssoci
where
Qparent=Domain (parent)
QmvAtt={Att: Attribute | Att.Ubound > 1}
QmvAssoci={ Associ: Association | Associ.Ubound > 1}

(b)

Figure 5.5: Query definitions

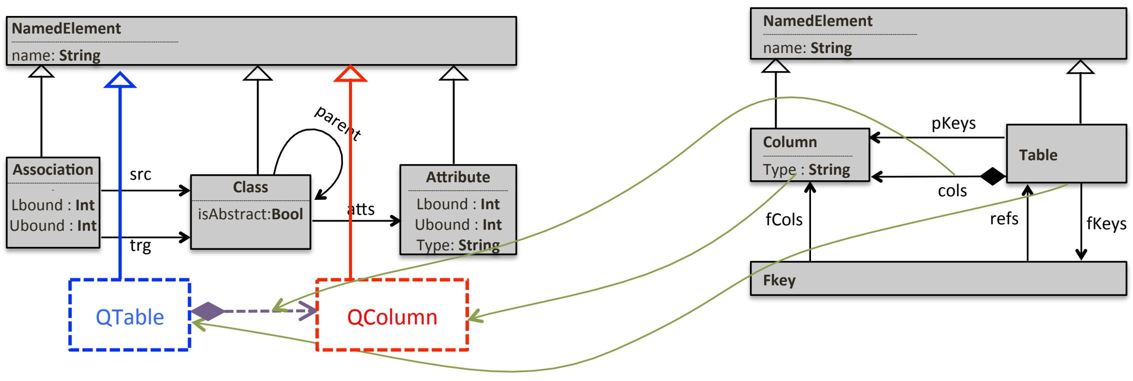


Figure 5.6: `QTable`, `QColumn`, and `Qcols` query represented abstractly

The blue dotted square named QTable in the right-hand side of Fig.5.6 abstractly represents the above QTable definition. After this query definition, the Table type in the target is mapped to this query (see the green arrow from Table to QTable in the same figure). Note that all the possible ways leading to the generation of the tables in the target are encapsulated within QTable, and this is implied by the Table to QTable mapping. The other parts of the transformation are similarly defined by identifying the required queries. For example, using the transformation specification, we need to figure out how the columns are generated, and continue to answer similar questions for all the other types (be it a square type or an arrow type) in the target metamodel. Therefore, based on the MT specification, we write a query like the one shown in Fig. 5.5(b) for the Column type and name it QColumn. As seen from the QColumn definition, it is also a disjoint union of a family of collections similar to QTable. Each of QmvAtt and QmvAtt collections (already defined in QTable) contributed twice in the definition expression. QsvAtt and QsvAssoci are the selections of single-valued attributes and single-valued associations, respectively, and Qparent is the domain of the parent association. The entire query is shown abstractly as a dashed block square labeled as QColumn in Fig. 5.7. Once again, since this query generates all the columns in the target metamodel, Column in the target is mapped to this query using a green arrow as seen in the figure.

Similar to node types in the target metamodel, arrow types should also be mapped to query results (or an original association) in the source. Hence we draw an arrow between the QTable and QColumn (see purple arrow called Qcols in Fig. 5.6) and associate a query definition to this arrow. The definition should express the relation between the elements of the QTable and QColumn. This is defined to be the union of all the arrows which are indexed from one to eight in Fig.5.5(c); restr(atts) is the restriction of the atts relation over the QsvAtt codomain; inv(src) is the inverse of the src relation, and the id arrows are the identity relations over their domains.

Continuing the above process to the other types in the target metamodel will bring us to the structure shown in Fig. 5.7. The entire figure shows the abstract definition of the ClassToTable MT in QueST. The new green elements in Fig. 5.7 called QFKey, QpKeys, Qrefs, QfKeys, and Qfcols are all new query results whose internal definitions are abstracted away in this picture. Note that there might be some intermediate queries that are composed to build the results in this figure. However, we did not present them here to keep the illustration simple. Further abstraction over Fig. 5.7 will provide us with the structure in Fig. 4.7 that represents the common structure in all QueST definitions.

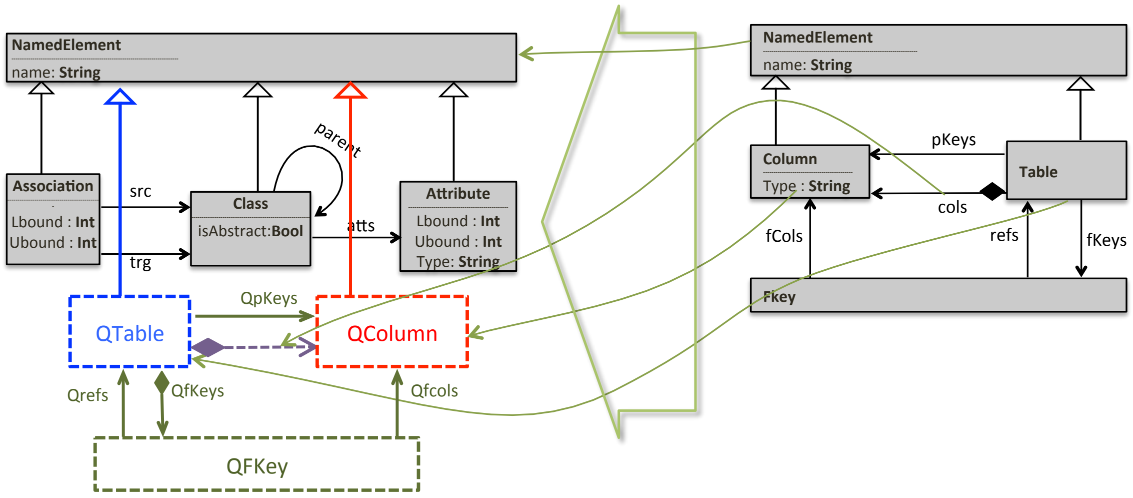


Figure 5.7: CD to DBS MT definition in QueST

### 5.2.4 ClassToTable in ETL, ATL, and QVT-R

The way a user thinks about the definition of an MT in rule-based approaches is as follows: she thinks about each element in the source and the way this element might contribute to the generation of elements in the target. We call the latter way of thinking the *source-to-target* paradigm. Each element in the source might affect the generation of multiple elements in the target. For example, for each class in the ClassToTable example, one table, one column and one foreign key should be generated in the target and also their corresponding references should be adjusted. Further, if a class inherits another class, there should be an additional column acting as a foreign key to model this relation between the corresponding generated tables.

In case multiple elements in the target model are generated from one specific element in the source model, there exist two options: either the generation of all those elements are handled within one rule, or it is divided into multiple rules. There exists no specific guideline as to which approach should be followed.

In the ETL implementation of the ClassToTable example, we follow the first option. The list of the rules and their corresponding specifications are presented in Fig. 5.8. As seen, the definition is divided between five rules; however, each contributes in the generation of more than one element in the target. The definitions of these rules are all provided in Appendix A.2. However, we have put the first rule implementation in Fig. 5.9 and briefly explain its content to illustrate its imperative nature.

In Fig. 5.9, line 3 defines an instance variable `class` that is considered an input of the rule. Lines 4–5 define two instance variables, `table` and `column`, that are considered outputs of the rule. That is, for every model element bound to `class`, two model elements of types `Table` and `Column` will be generated and will be bound to

| | ETL Rule | Description |
|---|---|---|
| 1 | **Class2Table** | For each class it generates a table and a column, which is also a primary key. If a class had a parent class, it generates another column as a foreign key referring to the parent class's corresponding table. |
| 2 | **mvAtt2Table** | For each multi-valued attribute, generates a table, two columns, and one foreign key. One of the columns is the value column and the other one is the foreign key column; the foreign key refers to the owner class's corresponding table. |
| 3 | **mvAssoci2Table** | For each multi-valued association, generates a table with two columns and two foreign keys, refereeing to, respectively, the source and to the target classes' corresponding tables. |
| 4 | **svAtt2Col** | For each single-valued attribute, generates a column. |
| 5 | **svAssoci2Col** | For each single-valued association, generates a column and a foreign key that refers to the target class's corresponding table. |

Figure 5.8: The list of rules of the ClassToTable example implementation in ETL

`table` and `column`, respectively. Lines 7–8 add the generated elements to the model container[1]. Lines 10–12 set the values for the target elements' properties. Lines 13 and 14 associate the generated column to the generated table as its column and as its primary key, respectively; in other words, these two lines generate instances of association types. The `if` condition at line 16 checks whether the `class` variable has a parent class; if so, it generates another column (line 17) and a foreign key (line 22), and sets the corresponding associations between them (see lines 17–25). Line 28 starts a for loop that iterates over the attributes of the class, and if any single-valued attribute is found, it adds the columns (corresponding to the attribute) to the list of columns of the generated table (see line 30). As lines 17 and 22 show, target elements can be generated inside the rules in ETL without being declared in the rule interface/signature. Allowing this makes it impossible to identify the rule functionality by just looking at a rule signature, and probably was a reason why it is prohibited in ATL matched rules.

There exists no drastic differences in the way we can organize the rules in ATL and QVT-R implementations in contrast to the ETL ones. However, as we mentioned earlier, ATL and QVT-R provide some declarative constructs that make the rule bodies relatively concise. The complete definition of the ClassToTable example in these two approach are provided in Appendices A.3 and A.4. However, we have provided the list of the rules for these implementations in Fig. 5.10 and Fig. 5.11. As seen from these lists, the number of the rules in the ATL and the QVT-R implementations are greater than the number of the rules in the ETL implementation. The reason is that we chose the rules in the ATL and QVT-R implementations to be more granular than ETL ones. For example, the `Class2Table` rule in the ETL definition is divided into two rules in the corresponding ATL definition (see lines 1–2 in Fig. 5.10). The `Class2Table` rule is even further divided into four rules in the the QVT-R implementation (i.e., see

[1]All the elements of the models should reside inside a container; this is a particular requirement just imposed by the implementation framework.

```
rule Class2Table
  transform
        class : myOO!Class
    to  table:myDB!Table,
        column:myDB!Column{

    dbschema.tables.add(table);
    dbschema.tables.add(column);

    table.name=class.name;
    column.name=class.name+"PK";
    column.type="INT";
    table.cols.add(column);
    table.pKeys.add(column);

    if (not (class.parent=null)){
        var col:new myDB!Column;
        col.name="parent-"+class.parent.name+"PK";
        col.type="INT";
        table.cols.add(col);

        var f:new myDB!Fkey;
        f.fCols.add(col);
        table.fKeys.add(f);
        f.ref=class.parent.getCorrTable();
    }

    for (a : myOO!Attribute in class.atts) {
        if (a.ubound = 1){
            table.cols.add(a.equivalent("svAtt2Col"));
        }
    }
}
```

Figure 5.9: An example of an ETL rule

| | Matched Rule | Specification |
|---|---|---|
| 1 | **Class2Table** | Similar to "Class2Table" rule in ETL. |
| 2 | **SubClass2Table** | |
| 3 | **mvAtt2Table** | Similar to "mvAtt2Table" rule in ETL. |
| 4 | **mvAssoci2Table** | Similar to "mvAssoci2Table" rule in ETL. |
| 5 | **svAtt2Col** | Similar to "svAtt2Col" rule in ETL. |
| 6 | **svAssoci2Col** | Similar to "svAssoci2Col" rule in ETL. |

Figure 5.10: The ClassToTable example rules in the ATL implementation

| | Relation | Specification |
|---|---|---|
| 1 | **Class2Table** | Similar to " Class2Table " rule in ETL. |
| 2 | ClassToCol | |
| 3 | **InherToCol** | |
| 4 | InherToFkey | |
| 5 | **mvAttToTable** | Similar to "mvAtt2Table" rule in ETL. |
| 6 | mvAttsToCol | |
| 7 | mvAttsToFkey | |
| 8 | **mvAssociToTable** | Similar to "mvAssoci2Table" rule in ETL. |
| 9 | mvAssociToCol | |
| 10 | mvAssociToFkey | |
| 11 | **svAttsToCol** | Similar to "svAtt2Col" rule in ETL. |
| 12 | **svAssociToCol** | Similar to "svAssoci2Col" rule in ETL. |

Figure 5.11: The ClassToTable example relations/rules in the QVT-R implementation

rules 1-4 in Fig. 5.11). Note that such finer decomposition was also possible in the ETL implementation; however, we just chose the list of rules differently in the three implementations to illustrate different possibilities of decomposing an MT definition in rule-based approaches.

## 5.3 Comparative Analysis

In previous sections, we explained implementations of two examples using ETL, ATL, QVT-R and QueST. In the following sections, using these examples, we analyze the following aspects of these approaches: 1) structural components, 2) their interrelations with metamodels, and 3) their composition mechanisms. By analyzing these aspects and sketching the differences between these rule-based approaches and QueST we explain the application of QueST in MT engineering from the following perspectives: declarativity, modularity, and incremental design. We also perform a white-box analysis to compare the way QueST queries are represented in the corresponding implementations in these rule-based approaches.

### 5.3.1 Structural Components

Rules are the main structural components of rule-based approaches. In contrast, queries are the main building blocks of the QueST approach. In the following, we sketch the differences between these two concepts.

#### Rules

The rules provide a mechanism to navigate the source model element-by-element and select them based on the provided properties. The target elements, accordingly, are created based on the selected source elements. In ETL, this mechanism is achieved by defining an instance variable of specific source type that is matched with a source model element. A guard expression restricts matching of this variable during a rule execution. For each matched element in the source, a corresponding instance is created in the target and is assigned to the target instance variable. In other words, the input(s) and output(s) of a rule are individual elements of, respectively, source and target models. For example, in Fig. 5.1, see the source element instance variable `person` at line 2, the guard expression at lines 6–14, and the target element instance variable `happyPerson` at line 3. When `person` is matched with an element of a source model, an instance of `HappyPerson` is created and assigned to the `happyPerson` variable. The variables are also used within the body of the rules (see lines 16–24).

In ATL and QVT-R, though the expressions in the rule bodies are relatively more declarative, the querying mechanism remains almost the same. For example, see the instance variables `person` (at line 7) and `happyPerson` (at line 11) in Fig. 5.2 that, respectively, refer to an instance of `Person` and an instance of `HappyPerson`. In the QVT-R implementation at Fig. 5.3, see the instance variables `person` at line 3 and `happyPerson` at line 7 that serve a similar purpose.

### Queries

Recall from Chapter 4 that the query operations in QueST are collection-wise operations; that is, their corresponding inputs and outputs are collections of elements rather than individual elements. For example, consider the Domain Selectionoperation (i.e., $Q_2$) used in the QueST HappyPeople example in Fig. 4.6; the query input is the relation `QBoth` and its output is a set (i.e., `QPerson`) and a relation (i.e., `isA`) –see the operation's explicit arity illustrated in Fig. 4.17.

### Comparison

QueST queries operate at a higher abstraction level than rules; that is, queries operate on collections while rules operate on elements of collections. Fig. 5.12 visualizes this difference between the rules and QueST queries. As seen, the queries' inputs are the metamodel elements, and their outputs are also metamodel elements. In contrast, rules' inputs and outputs are model elements. In this respect, we can say that QueST queries are a more declarative construct than rules in rule-based approaches.

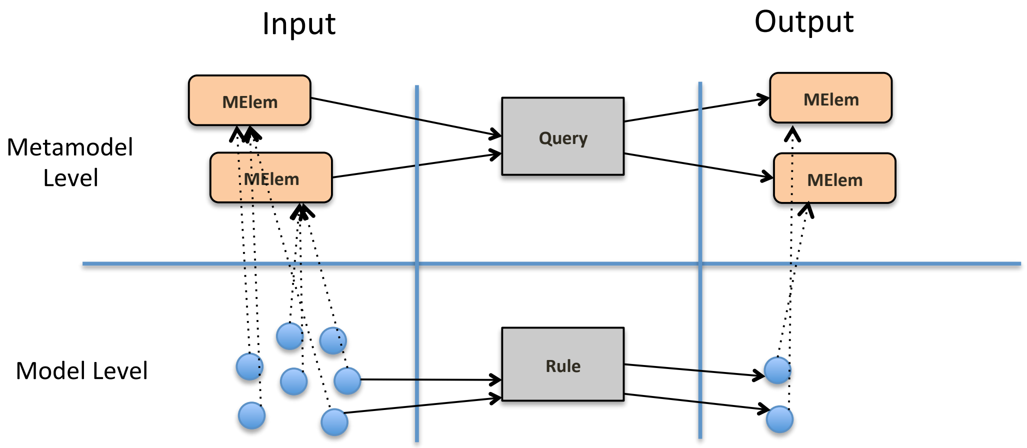


Figure 5.12: Queries Operate on a more higher level of abstraction.

The declarativity, in the sense of working on higher levels of abstraction, is already successfully applied in the database community. If we sketch an analogy between database schemas and metamodels, the way QueST queries work on the metamodels is similar to the way in which SQL queries work on the relational schema, and the way querying mechanisms work in the rule-based languages is similar to querying languages of Network (Taylor and Frank, 1976) and Hierarchical (Tsichritzis and Lochovsky, 1976) databases. The lower popularity of the Network and Hierarchical approaches in contrast to the great success of relational databases that uses the SQL language might be the good evidence of the benefits of declarative collection-wise manipulation of data over their element-wise processing. However, there is always a caveat of having to educate designers to think and write declarative queries.

### 5.3.2 Rules and Queries' Relation to Metamodel Types

In the following, we analyze how rules and QueST queries are related to the source and the target metamodel elements/types.

#### Rules

Examining the relations between the rules and the metamodel elements/types in the studied rule-based languages (i.e., ETL, ATL, and QVT-R) provides us with the model presented in Fig. 5.13. `SElem` and `TElem` in the figure represent, respectively, the source and the target metamodel types. On the left, the `defOn` relation specifies that each rule might be defined over one to many source metamodel types (see the 1..* multiplicity over `defOn`)[1]. The `usedIn` arrow is the converse of the `defOn` arrow. Its 0..* multiplicity indicates that each source type might be used in zero to many rules. For example, in the ATL ClassToTable example presented in Appendix A.3, the `Class` type is used in both `Class2Table` and `SubClass2Table`, besides some others.

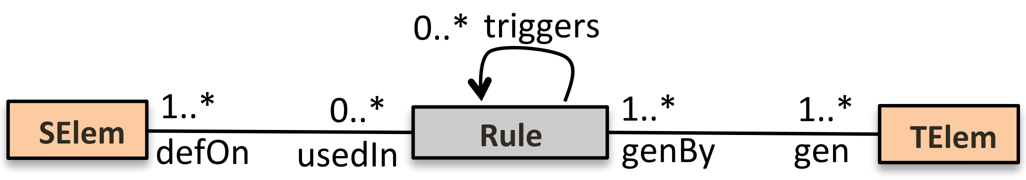


Figure 5.13: Rules' relations with source and target metamodel types.

On the right-side of the model in Fig. 5.13, the `gen` relation represents a rule contribution in the generation of instances of a target type. For example, the `mvAttToTable` rule in Fig. 5.9 generates instances from two different types `Table` and `Column`. The 1..* multiplicity over the `gen` relation indicates that a rule should be related to at

[1]In ATL and ETL, each rule can be defined over one and only one input type; however, in QVT-R, there is no upper limit for the number of the input types over which a rule can be defined. We left the multiplicity constraint to be 1..* to cover all scenarios.

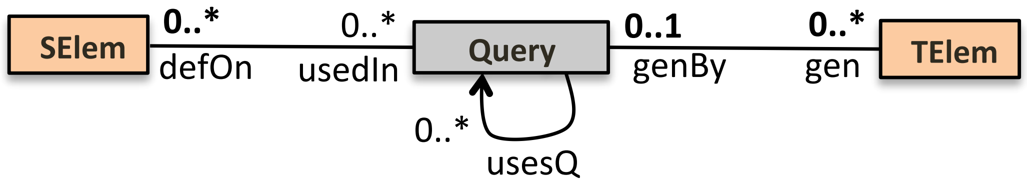


Figure 5.14: QueST queries' relations with source and target metamodel types.

least one target type. `genBy` is the converse of `gen`. Similarly, its 1..* multiplicity indicates that the instances of a target type should be generated by at least one rule. Note that there is no upper limit and the target type instances can be generated by any number of rules. For example, tables are generated by both `Class2Table` and `mvAtt2Table` in the ClassToTable rule-based implementations –see the description of these rules in Fig. 5.8. The looping arrow called `triggers` over the `rule` entity will be discussed in Sect. 5.3.3.

### Queries

We did a similar analysis as above over the QueST queries to model the relations of the queries with the metamodel elements. Fig. 5.14 summarizes the analysis result. Similar to Fig. 5.13, `SElem` and `TElem` represent the source and the target metamodel elements, respectively. The semantics of the arrows in Fig. 5.14 are the same as Fig. 5.13; however, their multiplicities differ. As seen in Fig. 5.14, queries do not need to use the source metamodel elements directly (see the 0..* multiplicity on `defOn`). For example, $Q_2$ in the HappyPeople example (see Sect. 4.2.1) uses a derived element `Qboth` as its input[1]. As the 0..* multiplicity of the `usedIn` arrow indicates, a source type might be used as an input for multiple queries. For example, `Person` is used as the input of both $Q_1$ and $Q_2$ in the HappyPeople example (see Sect. 4.2.1 again).

On the right-side of Fig. 5.14, the 0..* multiplicity of the arrow `gen` implies that all the queries do not need to contribute directly to the generation of the output elements. For example, the $Q_1$ output in Sect. 4.2.1 does not directly generate elements in the target model, so it is not related to any target type; however, it is used as an input of another query. The unrestricted upper-bound idicates that a query might have multiple outputs that each might contribute to the generation of the instances of variant types. We do not have such a query in our examples; however, in Fig. 4.6, if we change the view-mapping such that `HappyPerson` is mapped to `QPerson`, `Vehicle` is mapped to `Person`, and `drives` is mapped to `QisA`, then $Q_2$ will be such a query contributing to the generation of instances of both `HappyPerson` and `drives`.

[1]Note that `Person` and `Car` are also inputs of the $Q_2$ query.

The 0..1 multiplicity over the converse of `gen` (i.e., genBy) indicates that instances of a specific type can only be generated by at most one query. For example, all the tables in the `ClassToTable` example in Fig. 5.6 are generated from `QTable`. The reason for the zero lower bound is that some target types might be directly mapped to source types instead of query definitions. For example, in Fig. 4.6, `Vehicle` in the target metamodel is directly mapped to `Car` in the source metamodel.

### Comparison

The main differences between the two models in Fig. 5.13 and Fig. 5.14 originate from the multiplicity constraints as follows.

- The `gen` relation multiplicity constraint in Fig. 5.14 states that some queries in QueST do not directly generate target elements; these are *helper* queries that contribute to incrementally building target models. In contrast, all the rules should directly generate some target elements as the multiplicity of `gen` implies in Fig. 5.13.
- The queries in QueST do not require direct use of source types (see the `defOn` multiplicity in Fig. 5.14). In contrast, as seen in Fig. 5.13, it is mandatory that the rules in the rule-based approaches use at least one source type in their definitions.
- According to the `genBy` multiplicity in Fig. 5.14, all instances of a target type are generated by at most one query in QueST. In contrast, multiple rules might directly contribute to the generation of elements of a specific target type in the rule-based approaches, as seen in Fig. 5.13.

According to the first and the second points above, the queries in QueST can use derived elements/types —defined by other queries— instead of original source types. They also do not need to directly contribute to the generation of the instances of target types. These two points make it possible to break down complex queries into simpler intermediary queries and incrementally construct them. We will further discuss this aspect in Sect. 5.3.3.

The third point is an indicator of the target-to-source paradigm that QueST suggests in MT definitions: an MT designer takes a type form a target metamodel and thinks about constructing a query that generates all instances of this specific type. The third point also enforces a simple modularity structure based on the shape of the target metamodel and prevents the scattering of the related query definitions all over the different parts of an MT definition as we will see in Sect. 5.3.4.

### 5.3.3 Composition Mechanisms

In this section, we examine the composition mechanism that rule-based languages offer in contrast to the QueST approach's composition mechanism.

#### Rules

It is possible for a rule to trigger other rules' execution. The `triggers` arrow at Fig. 5.13 specifies this kind of relation between the rules. For example, the rules (i.e., relations) referenced within the *where* clauses of a QVT-R rule are triggered after the execution of the original rule. Or, it is possible that a rule in ATL calls other rules (called *helper* rules). Nonetheless, rules cannot be composed in the sense that the outputs of a rule are used as the inputs of another rule. For example, as the rules' specifications in Fig. 5.8 imply, all of the rules directly get their inputs from the source model and generate elements in the target model (see also the rules' definitions in Appendices A.2, A.3, and A.4).

#### Queries

It is possible in QueST to compose queries into larger queries or decompose them into smaller queries. For example, $Q_1$ and $Q_2$ in the HappyPeople example (see Sect. 4.2.1) can be composed into one query. The arity of this composed query is presented in Fig. 5.15. The query takes two arrows and generates a node and an arrow connecting the node to the domain of the input arrows. Thus, instead of defining two queries $Q_1$ and $Q_2$, we could have defined one (instance of the) query presented is Fig. 5.15.

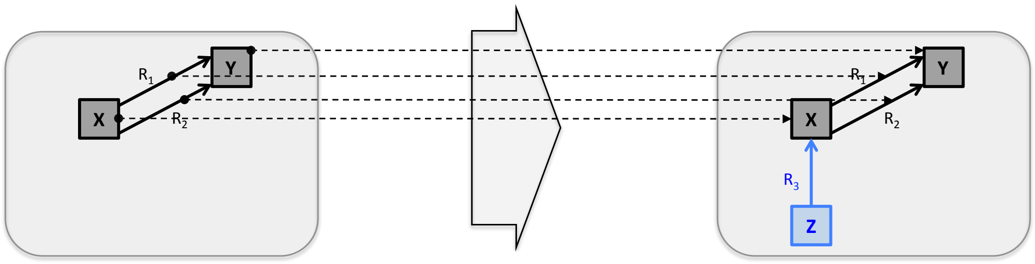


Figure 5.15: The arity of the composed query

In the reverse direction, complex queries can be decomposed into simple ones in QueST. For example, the query that defines `QTable` in Fig. 5.6 can be decomposed into three queries as presented in Fig. 5.16. In this figure, $Q_1$ selects single-valued associations; $Q_2$ selects single-valued attributes[1]; $Q_3$ aggregates elements from the latter two queries with the original `Class` type and builds `QTable`.

[1] $Q_1$ and $Q_2$ are, in fact, two instances of a single select operation with different select predicates.

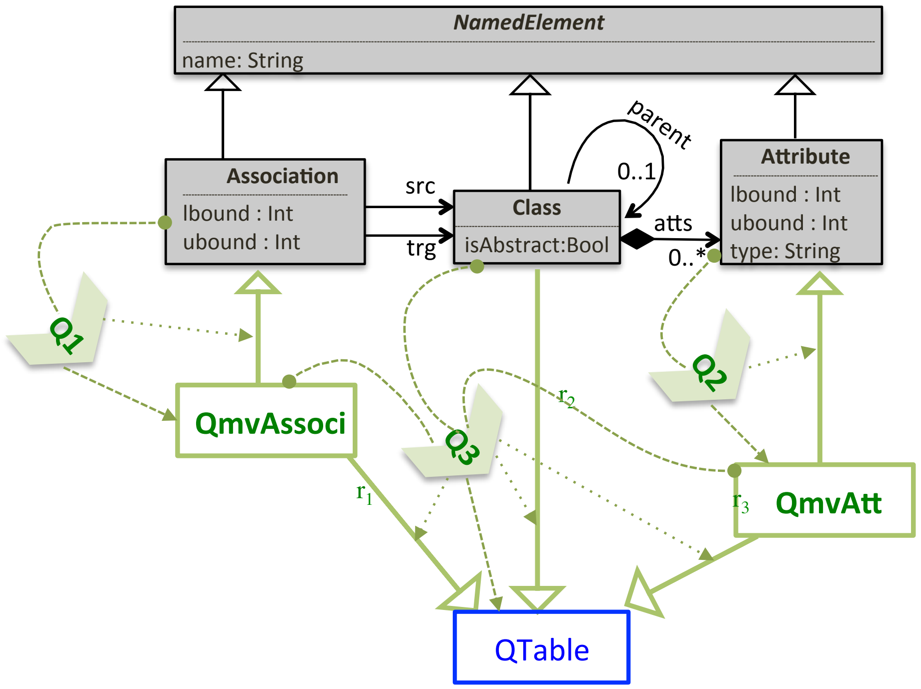


Figure 5.16: `QTable` decomposition into three queries.

## Comparison

In the following, we compare QueST and the rule-based approaches concerning the compositionality features that they provide for building MT definitions.

In the rule-based approaches, an MT definition can be arbitrarily distributed among different rules; that is, each rule takes responsibility for the generation of a portion of a target model. However, since rules do not allow chaining, this distribution is one dimensional. Fig. 5.17 might help communicate this one-dimensionality concept. In this figure, the chevrons represent the rules. The left and the right bar represent a source and a target model, respectively. The horizontal arrows from the source model to the rules indicate rules' inputs. The horizontal arrows from the rules to the target model indicate the rules' output. As seen, all rules are aligned vertically, each taking some responsibility in generating a portion of the target model; however, they cannot collaborate horizontally to incrementally build target elements. The reason is that the generation of a target element should occur in one step passing through only one rule. The dotted lines between the rules are instances of the `triggers` relation in Fig. 5.13 and not outputs of the rules passed to other rules.

In contrast, an MT definition in QueST can be distributed both vertically and horizontally among the queries. Fig. 5.18 illustrates this two-dimensional distribution. The interpretation of the elements in this figure is similar to Fig. 5.17, except the

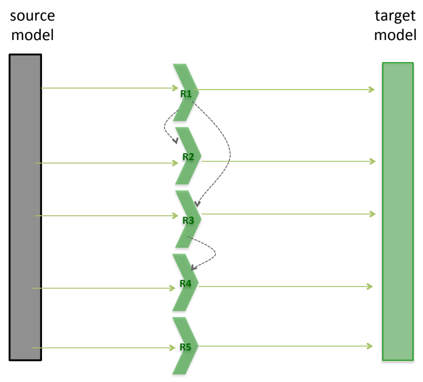


Figure 5.17: Rule orchestration in a rule-based MT definition.

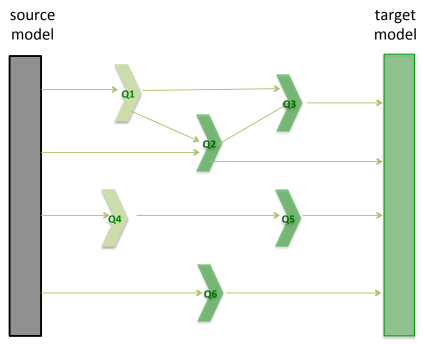


Figure 5.18: Query orchestration in a QueST MT definition.

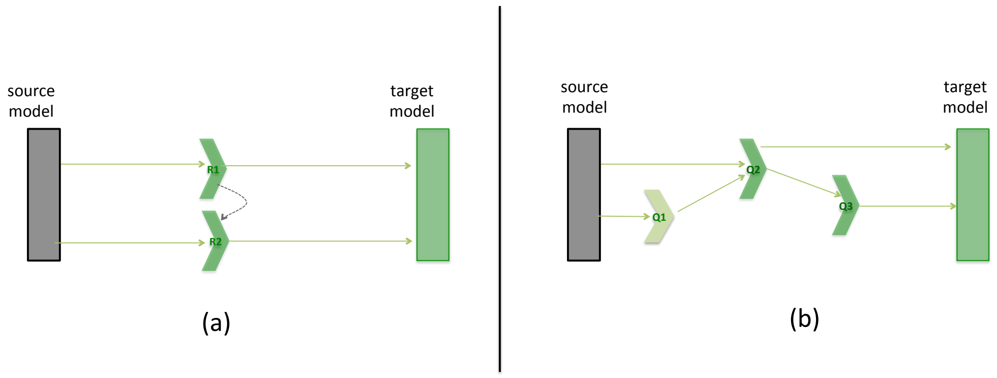


Figure 5.19: The Rules' vs. Queries' orchestration in the HappyPeople example.

chevrons representing queries rather than rules. As seen, the queries can receive inputs directly from the source model or other queries or both, and can serve as providers of inputs to other queries, or directly generate a portion of the target model. For example, $Q_3$ receives inputs from $Q_1$ and $Q_2$, and generates a portion of the target model. $Q_2$ receives its input directly from the input model, and, in addition to providing inputs to $Q_3$, generates a portion of the target model. The queries that do not directly contribute to the production of target model elements are faded in the figure to distinguish them from the other queries. Recall that these queries are called helper queries in QueST. $Q_1$ and $Q_4$ are helper queries in Fig. 5.18. Note that the arrows between the queries in Fig. 5.18 are instances of the `useQ` relation in Fig. 5.14. Fig. 5.19(a) and Fig. 5.19(b) present, respectively, the orchestration of the rules and the queries in the HappyPeople example according to the corresponding definitions in Sect. 5.1 and Sect. 4.2.1.

In a nutshell, the QueST two-dimensional pattern in structuring MT definitions is more flexible than the one-dimensional pattern in rule-based languages. The reason is that the complexity of the target model generation process can be divided into smaller queries not only horizontally but also vertically, so an MT definition can be achieved incrementally.

### 5.3.4 A White-box Analysis

In the previous sections, we compared the structural components of the rule-based approaches and the QueST approach by considering the corresponding components (i.e., rules and queries) as black-boxes. In this section, we perform a simple white-box analysis by looking inside these components and comparing the way QueST query definitions appear within the corresponding rule-based ones.

#### Analysis Explanation

We chose the ClassToTable example as the basis of our analysis. We carefully examined the rule contents in the ETL, ATL, and QVT-R implementations of this example and marked the places at which query definitions from the QueST ClassToTable implementation appear. We performed this for the three queries `QTable`, `QColumn`, and `Qcol` and employed certain notations with the following semantics: 1) The ■ signs are used to mark the parts corresponding to the generation of tables (the `QTable` query); 2) The ★ signs are used to mark the parts corresponding to the generation of columns (`QColumn` query), and 3) The ♦ signs are used to mark the parts which associate the columns to the tables (`Qcols` Query) — see the latter notations in Fig. 5.5. We further add numbers above these markings that will correspond to the numbers in Fig. 5.5(c) for each marking; for example, $\overset{2}{\blacksquare}$ refers to the `mvAtt` component of the `QTable`, and $\overset{3}{\blacklozenge}$ and $\overset{3}{\bigstar}$ refer to the `Qparent` arrow of `Qcols`, and the `Qparent` component of `QColumn`, respectively.

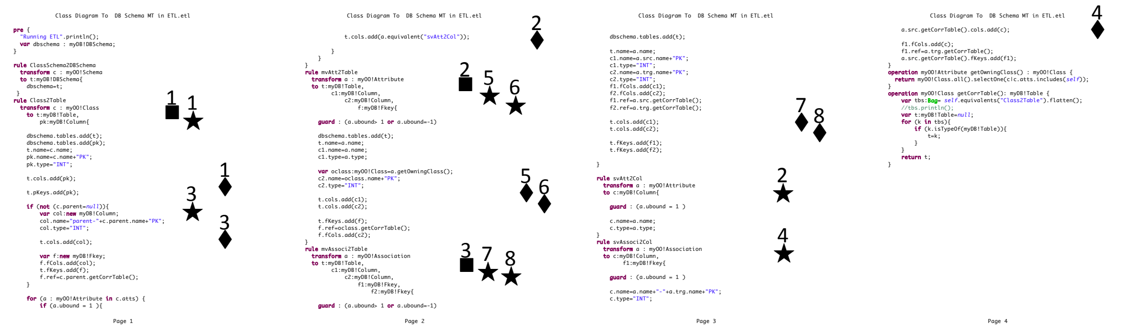


Figure 5.20: Query dispersal in ClassToTable ETL implementation.

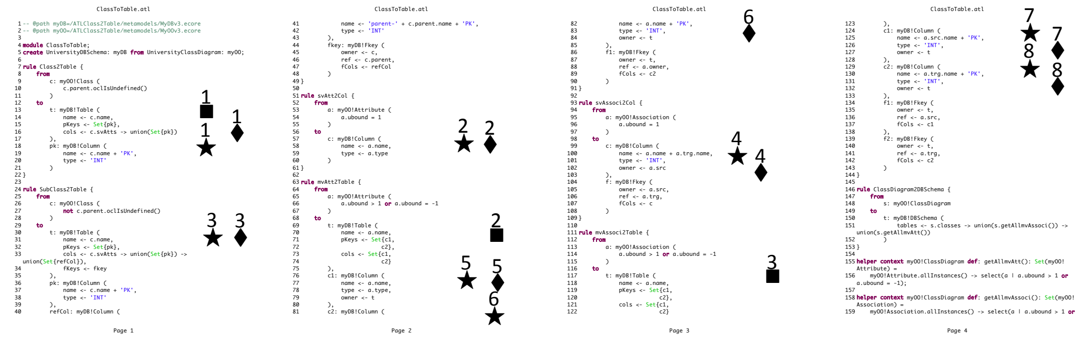


Figure 5.21: Query dispersal in ClassToTable ATL implementation.

To provide an abstract view of the marking results, we squeezed down the rule-based definitions by greatly decreasing their corresponding font-sizes. The results of the marking task performed over the ETL, ATL, QVT-R definitions are presented in Fig. 5.20, Fig. 5.21, and Fig. 5.22, respectively. As these figures show, each marking sign is dispersed over the entire code in all of the three definitions. For example, the eight components of `Qcols` which are indexed from one to eight are scattered throughout the entire definition and among the different rules in all of the three implementations. This means that the different components of a QueST query in Fig 5.5 are dispersed all over the corresponding definitions in the rule-based implementations. Repeating the above experiment for the other queries in Fig. 5.7 reveals a similar dispersal of their corresponding definitions over the entire code in the rule-based implementations[1].

[1]We did not put all of these markings in the figures to avoid cluttering the illustrations.

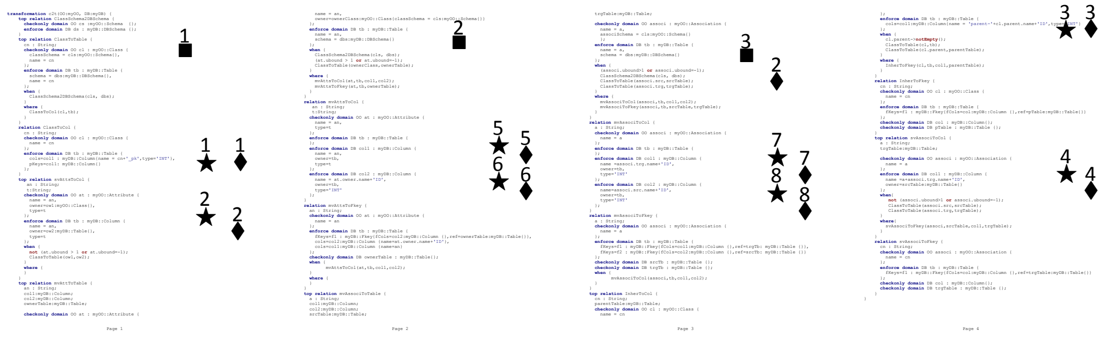


Figure 5.22: Query dispersal in ClassToTable QVT-R implementation.

## Rationale Behind Query Dispersal

It might be argued that what are presented in this thesis, as the definitions of the ClassToTable example in ETL, ATL, and QVT-R are subjective, in the sense that there might be different implementations for the ClassToTable example using these languages, such that the demonstrated query dispersals are prevented. We believe that the scattering of the QueST queries would happen in any implementation, because of the three reasons briefly discussed below:

- **One to many and many to many relations.** In Sect. 5.3.2, we analyzed the structural components of the rule-based languages in relation to source and target metamodel elements. We explained that, in the rule-based languages, the instances of a target type are generated via different rules; that is, one target metamodel element can be referenced in many rules/relations in a transformation definition. This makes the queries generating the instances of a specific type spread between different rules/relations.

- **Arrows are secondary elements.** References to arrows in MT definitions in ETL, ATL, and QVT-R happen by means of nodes; rules are defined primarily for the nodes, and arrow definitions are implicit inside these rules. More concretely, it is not possible to define an arrow as a target of a rule in ETL and ATL, or as a target domain of a relation in QVT-R. This means that if the queries generating nodes are dispersed, then the queries for generating the arrows which are referenced by these nodes will also be dispersed. The ⧫ marking signs appearing everywhere close to the ★ signs in Fig. 5.20-5.22 illustrate this phenomenon.

- **Flattening the graphical structure.** ETL, ATL, and QVT-R are textual languages, while as it may be seen from Fig. 5.7, MT definitions contain graphical constructs. A representation of a graphical construct in a textual format causes a scattering of references to the graphical elements, inside the representation; for example, a node with several incoming edges in a graph would be inevitably referenced in different places in the graph's textual representation.

## 5.4 Conclusion

In this chapter, we compared QueST and rule-based approaches (i.e., ETL, ATL, and QVT-R) from the following three perspectives: 1) the main structural components of the approach, 2) the components' relations to the metamodel elements, and 3) the components' composition mechanisms.

**Structural Components.** Rules are the main components of the MT definitions in the rule-based languages. In contrast, queries are the main components in the QueST definitions. We showed that rules process models element-by-element, while queries process them as collections. In other words, rules are defined at the *model level*, while queries are defined at the *metamodel level*.

**Control Flow Complexities.** The query definitions in QueST hide the implementation details (i.e., control flow structures) under the hood of declarative queries. However, the control flow structures appear either inside the rule definitions or as triggering dependencies between the rules in the rule-based approaches. As an analogy, similar to the Join/Union operations in relational databases that hide the underlying implementation control flows, the declarative queries in QueST abstract away control flow implementation details.

**Thinking Paradigm.** The thinking paradigm in the rule-based approaches is *source-to-target*; that is, the user thinks about the way each source *model element* affects the generation of the elements in the target model. In contrast, the thinking paradigm in QueST is *target-to-source*; that is, the user thinks about the queries that generate elements of a specific *metamodel element* in the target.

**Arrows Are Secondary.** Arrows/associations in the rule bases approaches are treated as secondary elements; that is, there cannot exist a rule whose input/output parameters are arrows. In QueST, arrows are treated as first-class citizens. Each query can accept arrows as its input/output parameters, in a similar way as it could accept the nodes. In fact, the inputs/outputs of queries in QueST are diagrammatic structures that include both nodes and arrows.

**Incremental MT Definition.** The outputs of one rule cannot be used as inputs of another rule. In this sense, there exists no cooperative data flow between the rules. In QueST, the outputs of queries might be bound to the inputs of other queries and construct a chain of collaborative query definitions. This enables users to define transformations incrementally in QueST.

**MT Design Pattern.** The rule-based approaches do not provide any specific guideline for the decomposition of the transformation definitions into rules. However, a specific blueprint (i.e., the target metamodel structure) always guides the decomposition and the arrangement of the queries in any QueST MT definition.

# Chapter 6

# Analysis of QueST MT Definitions

"Program testing can be used to show the presence of bugs, but never to show their absence."

Edsger Dijkstra

There exist two main approaches to verifying whether an MT definition satisfies the requirements of a transformation. One way is to define test cases and run these tests against the MT definition. Another way is to use mathematical tools and techniques to check the properties of the MT definition. Using testing techniques to verify MT definitions is quite straightforward and follows almost the same rules and techniques used in testing ordinary software applications. In this respect, QueST MTs can be treated similarly to other MT definitions. However, employing mathematical techniques to verify an MT definition has a prerequisite: the existence of a mathematical structure/specification corresponding to the MT definition. Since QueST is based on formal foundations, MT definitions in QueST have such corresponding mathematical structure that enables property checking of MT definitions to be carried out mathematically, besides the testing techniques.

In this chapter, we will demonstrate that each MT in QueST is equivalent to a logical theory and checking properties for an MT is equivalent to analyzing correctness of such properties (i.e., demonstrating that they are theorems) in its corresponding theory. We will explain the idea by encoding the underlying theories of the two examples used before (i.e., HappyPeople and ClassToTable) as Alloy (Jackson, 2002) specifications; then, we will define and analyze some properties over them to demonstrate the way MT analysis can be carried out.

The basic idea of our approach to property checking is as follows: an MT definition $\boldsymbol{D}$ is encoded as a logical theory $\boldsymbol{Th}(\boldsymbol{D})$ in an appropriate logic; a property to be checked is encoded as a logical proposition (sentence) $\boldsymbol{P}$, and checking the property for $\boldsymbol{D}$ amounts to checking the validity of the semantic entailment $\boldsymbol{Th}(\boldsymbol{D}) \models \boldsymbol{P}$, which means that all execution instances of $\boldsymbol{D}$ satisfy the property. For such a semantic validity check, we can use a model checker or an instance generator like Alloy (for a limited scope analysis). If the logic used for encoding is complete, we can replace semantic validity by syntactic provability, $\boldsymbol{Th}(\boldsymbol{D}) \vdash \boldsymbol{P}$, so the problem would be equivalent to proving or disproving the correctness of the proposition $\boldsymbol{P}$ in the theory $\boldsymbol{Th}(\boldsymbol{D})$. This process can be performed manually, or by receiving some assistance from a theorem prover. We have not used a theorem prover; however, we exploited the Alloy analyzer in refuting correctness of some defined properties. We also provided manual proofs for the properties we defined over the examples used.

This chapter is organized as follows. First, we will provide an introduction to the Alloy language explaining its syntax and semantics to keep the chapter self-contained. Then, we will discuss the encoding techniques used in the encodings of different parts of a QueST MT. We will present the encodings of the two running examples we used in the previous chapters (i.e., ClassToTable and HappyPeople) as Alloy theories/specifications. Following this, we will define some properties over these MT definitions. We will analyze the properties against the MTs by running the Alloy analyzer over the corresponding specifications, and manually prove the ones whose correctness the analyzer cannot refute. Finally, we will conclude the chapter.

## 6.1 Alloy Background

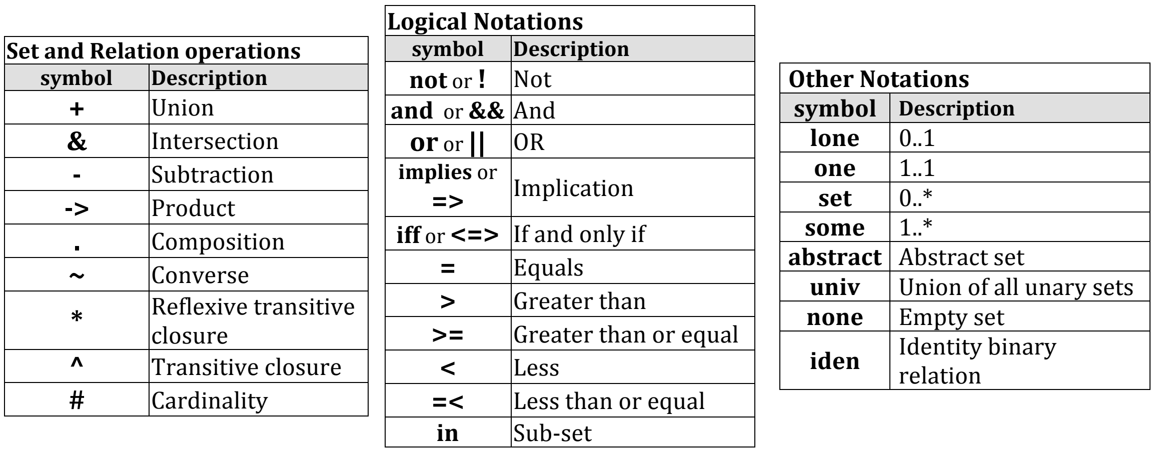
Alloy (Jackson, 2012) is a formal language developed at MIT (Jackson, 2002) and released along with a set of tools that together allow the building of the models of systems and the analysis of their properties. The convenience of using the Alloy tools and the language's intuitive syntax were the main reasons we chose Alloy as the host language for the QueST MT encodings. Alloy's underlying language is relational first-order logic with certain additional features that increase its expressive power (Jackson, 2002). Its particular relational aspect is inspired by the Z language (Spivey and Abrial, 1992) and Tarski's relational calculus (Tarski, 1941).

Alloy is not primarily built for encoding transformations; thus, encoding an MT in the language needs to exploit some techniques and applying some workarounds that will be explained in the following sections. However, before then, we will briefly introduce the language constructs.

### 6.1.1 Sets and Relations

Alloy's primary concepts are sets and (binary) relations, so that any system is modeled as a collection of sets and relations with a number of constraints defined over them. Relations are understood both *extensionally* (as sets of ordered pairs, $r \subseteq A \times B$)

and *navigationally* as partial multi-valued mappings (thus, we might use $r : A \nrightarrow B$ instead of $R : A \times B$ to give a relation declaration). The major operations of the extensional view are ordinary set union, intersection, subtraction, and cartesian product. The major operation of the navigational view is relation composition: the relations $r_1 : A \nrightarrow B$ and $r_2 : B \nrightarrow C$ can be composed to build the relation $r_1.r_2 : A \nrightarrow C$ with the following semantics: $\forall a : A, \forall c : C, (a, c) \in r_1.r_2 \iff \exists b : B, (a, b) \in r_1 \wedge (b, c) \in r_2$. Alloy provides these relational algebraic operations as well as the operation of transitive closure. Moreover, Alloy provides notations for writing FOL formulas with logical connectives and quantifiers. The left-most table in Fig. 6.1 provides a concise list of standard Alloy notations and their corresponding semantics for sets and relations.



**Set and Relation operations**

| symbol | Description |
|---|---|
| **+** | Union |
| **&** | Intersection |
| **-** | Subtraction |
| **->** | Product |
| **.** | Composition |
| **~** | Converse |
| * | Reflexive transitive closure |
| **^** | Transitive closure |
| **#** | Cardinality |

**Logical Notations**

| symbol | Description |
|---|---|
| **not** or **!** | Not |
| **and** or **&&** | And |
| **or** or **\|\|** | OR |
| **implies** or **=>** | Implication |
| **iff** or **<=>** | If and only if |
| **=** | Equals |
| **>** | Greater than |
| **>=** | Greater than or equal |
| **<** | Less |
| **=<** | Less than or equal |
| **in** | Sub-set |

**Other Notations**

| symbol | Description |
|---|---|
| **lone** | 0..1 |
| **one** | 1..1 |
| **set** | 0..* |
| **some** | 1..* |
| **abstract** | Abstract set |
| **univ** | Union of all unary sets |
| **none** | Empty set |
| **iden** | Identity binary relation |

Figure 6.1: List of Alloy notations and their meaning.

Unary sets are declared using the `sig` keyword. For example, a set A is declared as follows.

```
sig A{}
```

Binary or n-place relations are declared inside curly brackets coming after their corresponding domains; for example, the following code declares two relations $r1 : A \times B$, and $r2 : A \times B \times C$.

```
sig A{r1:B, r2:B->C}
```

## 6.1.2 Constraints

Constraints are defined within *fact* blocks. For example, the following axiom

**Axiom 1 (invIsTotal).** $\forall b : B, \exists a : A, (a, b) \in r_1$

that constrains the converse of the relation $r_1 : A \times B$ to be total is specified in Alloy as follows:

```
fact { all b:B | some a:A |  a->b in r1}
```

There exist certain predefined concise notations for frequently used constraints such as cardinality constraints. The right-most table in Fig. 6.1 lists these concise notations and their meaning. For example, the following `lone` notation requires the relation `r1` to be a partial function.

```
sig A{r1:lone B}
```

Another example is adding the `one` modifier in front of a set declaration that makes it a singleton, as seen in the following.

```
one sig A{}
```

Until now, the constraints are specified using first-order logic notations (for example, the `forall` and the `some` notations are used). The flexibility of the Alloy language allows specifying constraints in other styles (Jackson, 2012, pp. 34). For example, Axiom 1 (invIsTotal) can be alternately expressed in the *navigational style* in Alloy as follows:

```
fact { all b:B | some b.~r1}
```

Even a pure *relational algebraic* style (Tarski, 1941) is supported in Alloy. Again, Axiom 1 (invIsTotal) could be rewritten in relational algebraic style as follows:

```
fact {univ.r1=B}
```

Logical operators and connectives in Alloy have very intuitive notations. The middle table in Fig. 6.1 lists these notations and their corresponding interpretations.

### 6.1.3 Analysis Features

Alloy provides two primary mechanisms to analyze a specification: 1) *assertions*, 2) *predicates*. In the first method, Alloy searches for a counterexample to the specified assertion. In the second method, Alloy attempts to find a model for the specification that satisfies the specified predicate.

Assertions are defined within *assertion blocks*. For example, in the following, it is asserted that each *attribute* belongs to one and only one *class*:

```
sig Class{atts: set Attribute}
sig Attribute{}
assert eachAttributesBelongToOneClass
  {all a:Attribute | one a.~atts}
```

Predicates are defined within *pred* blocks. For example, the above assertion expression is written within a predicate block as follows:

```
pred allAttAreInAClass{all a:Attribute | one a.~atts}
```

For further information regarding the employment of each mechanism (i.e., whether to use assertions or predicates) during a system design and analysis, please refer to Jackson (2012).

### 6.1.4 Scope Parameter

Alloy specifications can be analyzed using the Alloy analyzer tool. The tool uses scope parameters whose values are implicitly or explicitly provided during an analysis task. The scope parameters determine the maximum or the exact cardinality of unary sets (i.e., signatures) that are used in verifying the underlying specifications. The default scope value is four; that is, the maximum cardinality of all the unary sets in a specification is assumed to be four. However, it is possible to customize this value and increase or decrease it either for all the sets or exclusively for some of them in an analysis run. As an analysis of a specification is always limited to the specified scopes in Alloy, there exists no certainty that the analysis results would hold true when we increase the scope boundaries.

Although the scope limitation might seem very restrictive at first, limited scope analysis is shown to be very helpful in design and development of the systems. The reason is that "most bugs have small counterexamples and if an assertion is invalid, it probably has a small counterexample". This last quoted statement is referred to as the *Small Scope Hypothesis* (Jackson, 2012). However, to prove that an assertion for an unlimited scope is also valid, we need to employ mathematical proofs as we will see in Sect. 6.4. Note that the inability to provide automatic proofs to the correctness of the assertions beyond the scope boundaries is not the limitation of Alloy per se, as languages like first-order logic are generally undecidable.

### 6.1.5 Facts

In this chapter, we will use the following two facts.

**Fact 1.** *Let $R_i : A_i \nrightarrow B_i$, $i = 1, 2$ be two relations, and $f : A_1 \rightarrow A_2$, $g : B_1 \rightarrow B_2$ two functions such that the following* commutativity *condition holds: $R_1.g = f.R_2$. Then there is a uniquely defined function $f * g : R_1 \rightarrow R_2$ between relations as sets such that for a pair $(a, b) \in R_1$, $f * g(a, b) = (f(a), g(b))$. Moreover, if functions $f, g$ are bijections, function $f * g$ is a bijection as well (and in this case, we say that relations $R_1$ and $R_2$ are isomorphic).*

*Proof.* Straightforward. □

**Fact 2.** *Tow relations satisfying the conditions of Fact 1 have the same head and tail multiplicities.*

*Proof.* Straightforward. □

Fact 1 will be used in the encoding of the view-mappings (see Sect. 6.2.2). Fact 2 will be used in proving Theorem 6.4.1 at page 115.

## 6.2 Encoding the HappyPeople Example

Before we start discussing the encoding techniques, let us fix some terminology. The term "model" is sometimes used to refer to an Alloy specification in the Alloy terminology. We already used this term to refer to an instance of a metamodel in MDE. Thus, when we use *model*, we mean a metamodel instance by default unless it is clear from the context that an Alloy specification is intended.

Recall from Chapter 4 (see Fig. 4.7) that a QueST definition consists of three major components: 1) The source and the target metamodels, 2) The queries augmenting the source metamodel, 3) The view-mappings associating the target types to the types in the source or in the augmented source metamodels. In the following, we explain the way we encode each of these components in Alloy.

### 6.2.1 Encoding Metamodel

Alloy's view of the world as consisting of sets and relations perfectly matches the understanding of metamodels in QueST[1]. Therefore, the source and the target metamodels of an MT can be directly encoded in Alloy. In Fig. 6.2, lines 3,5 and 6 encode the source metamodel, and lines 9 and 11 encode the target metamodel (see the metamodels' spec in Sect. 4.1.1). Classes become signatures/sets, and associations become relations defined inside curly brackets after their corresponding domain signature; e.g., $likes : Person \nrightarrow Car$ and $owns : Person \nrightarrow Vehicle$ are two relations defined under signature *Person*. The keyword *set* in these relations' definitions specifies the multiplicity constraints 0..*. Line 6, specifies the multiplicity constraint 1..* at the left-side of the *owns* relation in the source metamodel (see Fig. 4.1). The two encoded relations –*map1* and *map2* at lines 10 and 12– are parts of the view mapping links explained below; they are not parts of the target metamodel encoding.

[1]In the encoding of QueST MTs, we assume set/relation interpretations of metamodels (see Sect. 4.4).

```
open util/relation
//------ Source metamodel -------------
sig Person {likes,owns:set Car,
  Qboth:set Car}    //query definition
sig Car{}
fact{ all c:Car | some p:Person | c in p.owns}

//------ Target metamodel and view-mapping ---------
sig HappyPerson {drives:set Vehicle,
  map1: QPerson}  //the mapping
sig Vehicle{
  map2: Car}  //the mapping

//the view-mapping constraints
fact{
  //map1
  bijection[map1,HappyPerson,QPerson]
  //map2
  bijection[map2,Vehicle,Car]
  //commutivity constraints ensuring map3 (see Fact 1)
  drives.map2=map1.Qdrives
}

//------ Query definitions  -------------
//Relation Intersection Query
fact {Qboth = likes & owns}

//Domain Restriction Query
sig QPerson in Person{ Qdrives :set Car }
fact {QPerson =  Qboth.univ }
fact { Qdrives = Qboth }
```

Figure 6.2: An encoding of the HappyPeople QueST example in Alloy

### 6.2.2 Encoding Queries

We encode each query operation by adding to the Alloy specification a family of sets and relations produced by the query. Recall from chapter 4 (see section 4.2.1), that we have defined three queries for the HappyPeople example. The *Relation Intersection* Query ($Q_1$) produces the relation `Qboth`. The *Domain Selection* Query ($Q_2$) produces the set `QPerson` and the relation `QisA`, and the *Arrow Composition* Query ($Q_3$) produces the `Qdrives` relation. In Fig. 6.2, see line 4 that defines `Qboth` and line 29 that defines `QPerson` and `Qdrives`. Note that the `QisA` relation in Fig. 4.6 is encoded implicitly by the *in* keyword at line 29, denoting the subsetting relation between `QPerson` and `Person`.

The semantics of the query operations are encoded as constraints. For $Q_1$, line 26 ensures straightforward intersection semantics for `Qboth`. Lines 30–31 specify the corresponding constraints for the $Q_2$ and the $Q_3$ results; `QPerson` = `Qboth`.*univ* at line 30 makes `QPerson` equal to the projection of `Qboth` onto `Person`, and `Qdrives` = `Qboth` at line 31 is, in fact, a concise encoding of `Qdrives` = `QisA`·`Qboth`, since `QisA` is an identity relation.

### 6.2.3 Encoding Mappings

In the HappyPeople example, the *view mapping* consists of three mapping links or just *maps* connecting the target metamodel with the source (augmented) metamodel. Semantically, if a map links two classes (i.e., node types, like *map1* and *map2*), then it specifies a bijection between the corresponding sets, and we add to our Alloy specification the corresponding facts (see lines 16–19 in Fig. 6.2). If a map links relations (i.e., arrow types, like $map3$), semantically it means a bijection between the relations as sets of pairs, and moreover, the following *commutativity* constraint: if $map3(h, v) = (p, c)$, then $map1(h) = p$ and $map2(v) = c$. However, Alloy does not allow us to define a relation between relations like *map3*, and we need to find a workaround. Fact 1 (see Sect. 6.1.5) helps here: the commutativity condition in this fact implies the existence of the required mapping between associations, and adding it to the Alloy spec (see line 21 in Fig. 6.2) completes the encoding of the view-mapping. In this way, the entire Alloy specification in Fig. 6.2 encodes the HappyPeople example in Fig. 4.6.

## 6.3 Encoding the ClassToTable Example

In the previous section, we explained the encoding techniques to specify the definition of the HappyPeople example in Alloy. We can use the same techniques to encode the

ClassToTable example (or any other QueST MT) in Alloy. If we assume the view-mappings to be bijective[1], we can simplify the encoding process as follows. We can avoid the encoding of target metamodels and view-mappings and, instead, identify the target metamodel with the view-mapping image. For example, as seen in Fig. 5.7, the MT view-mapping is bijective; the `Column` type in the target metamodel is mapped to the derived type `QColumn`. Thus, we will just encode `QColumn` and skip the encoding of `Column` and the view-mapping link between `Column` and `QColumn`, consequently. This does not cause any problem in verifying properties (see Remark 2 at page 116)

## 6.3.1 Encoding Metamodel

Fig. 6.3 presents the encoding of the ClassToTable example source metamodel in Alloy. Lines 5–11 define the nodes/sets and arrows/relations, and lines 14–28 define the constraints on the metamodel. Line 5 defines the signature `Str` to model strings. Alloy does not provide support for the *string* type, so it should be modeled similarly to other entities. The `abstract` keyword at line 6 makes `NElement` abstract; that is, no direct instance of this node can exist. The rest of the encoding notations are self-descriptive and are similar to what were explained in the encoding of the HappyPeople metamodels. The constraint `diamond` (lines 14–16) ensures that each attribute belongs to one and only one class. The constraint `noLoop` (lines 17–19) ensures that the inheritance association (i.e., `parent`) is not circular. The caret (ˆ) symbol at line 18 is the transitive closure operation (see Fig. 6.1). The constraints defined inside the `bounds` block (lines 20–28) are the ones regarding the boundary properties (`lbound` and `ubound`) of the `Attribute` and `Association` nodes in the source metamodel. For example, line 21 ensures that the lbound (lower-bound) value for the attributes is always greater than or equal to 0.

## 6.3.2 Encoding Queries

Similar to the HappyPeople example, queries are encoded by adding a family of sets and relations produced by the queries. We will only explain the encoding of the query that defines `QTable` in the ClassToTable example. The rest of the queries are similarly encoded and their encodings are provided in Appendix A.1.

Recall from Sect. 5.3.3 (see Fig. 5.16), that we decomposed the definition of `QTable` into three queries: $Q_1$ defines `QmvAssoci`, $Q_2$ defines `QmvAtt` and $Q_3$ combines the two elements (i.e., `QmvAssoci` and `QmvAtt`) and `Class` to construct `QTable`. $Q_1$ and $Q_2$ are *select* queries. In Fig. 6.4, $Q_1$'s output (i.e., `QmvAssoci`) is encoded at lines 11–15. $Q_2$'s output (i.e., `QmvAtt`) is encoded at lines 4–8. The arrows connecting `QmvAtt` to

[1]In this thesis, we assume the view-mappings are bijective; however, for any MT definition with a non-bijective view-mapping, it is straightforward to convert it to a one with a bijective mapping.

```
//*****************************************************
//-------------------Source metamodel-------------
//*****************************************************
// ---metamodel nodes and arrows---
sig Str{}
abstract sig NElement {name: one Str}
sig Class extends NElement
  {parent :lone Class, atts: set Attribute}
sig Association extends NElement
  {lbound, ubound: one Int, src, trg: one Class}
sig Attribute extends NElement{lbound,ubound:one Int}

// ---constraints-------------------
fact diamond{
  all a:Attribute | one c:Class | a in c.atts
}
fact noLoop{
  all c:Class | not (c in c.^parent)
}
fact bounds{
  all a:Association | a.lbound >=0
  all a:Association | not (a.lbound =0 and a.ubound =0)
  all a:Association | (a.lbound <= a.ubound) or (a.ubound=-1)

  all a:Attribute | a.lbound >=0
  all a:Attribute | not (a.lbound =0 and a.ubound =0)
  all a:Attribute | (a.lbound <= a.ubound) or (a.ubound=-1)
}
```

Figure 6.3: An encoding of the ClassToTable source metamodel in Alloy

```
// ---Helper queries to build QTable
// --QmvAtt1 definition----------------
//****************************
sig QmvAtt1 in Attribute{}
fact {
  all a:Attribute |
    (a.ubound =-1 or a.ubound >1) iff a in QmvAtt1
}
// --QmvAssoci1 definition------
//****************************
sig QmvAssoci1 in Association{}
fact {
  all a:Association |
    (a.ubound =-1 or a.ubound >1) iff a in QmvAssoci1
}
//****************************
// -- QTable query:
// --   the Table node in target metamodel is mapped to QTable
//****************************
sig QTable extends NElement {
    r1: set QmvAssoci1,
      r2: set Class,
        r3: set QmvAtt1,
          a1,a2,a3,a4,a5,a6,a7,a8,Qcols, QpKeys: set QColumn,
            k1,k2,k3,k4,k5,QfKeys: set QFKey

}
fact {
  bijection[
      QTable<:r3+QTable<:r1+QTable<:r2,
        QTable,
          QmvAtt1+QmvAssoci1+Class]
}
// -- this is to populate the name property of QTable elements.
fact {
  all q:QTable | q.r1!=none  => q.name=q.r1.name else
      q.r2!=none => q.name=q.r2.name else
        q.r3!=none => q.name=q.r3.name
}
```

Figure 6.4: An encoding of the QTable query in Alloy

`Attribute`, and `QmvAssoci` to `Association` in Fig. 5.16 are inclusion mappings and are both encoded with the `in` keyword in Fig. 6.4 at lines 4 and 11, respectively. The select predicates for $Q_1$ and $Q_2$ are encoded inside the fact blocks (see lines 13–14, and lines 6–7). `QTable` is encoded between lines 20 to 39. The constraint inside the first `fact` block (lines 29–32) is the encoding of $Q_3$'s semantics. The second fact block (i.e., lines 36–38) defines the name property of `QTable`. The complete encoding of the ClassToTable example in Alloy can be found in Appendix A.1.

## 6.4 Analyses of MT Definitions

Sometimes the user is interested to ensure that a particular property is true for an MT definition, and sometimes she is interested to check if a particular property does not hold. We will examine both of these scenarios in this section. Having a corresponding logical interpretation of an MT definition enables us to formulate these concerns as propositions. We will use the Alloy analyzer to verify such propositions over the two encoded MT definitions (i.e., HappyPeople, and ClassToTable). For each encoded proposition, the analyzer attempts to find counter examples to refute the validity of the proposition. If it fails, there is a chance that the proposition is valid; however, a proof should be provided. We will provide manual proofs for such cases. We first analyze the HappyPeople example and then the ClassToTable example.

### 6.4.1 Analyzing the HappyPeople Example

Recall the HappyPeople example is the translation of the people who like and own cars, to people who are happy in case they both like and own the same vehicle. Suppose we want to verify the correcness the following three input-output properties over this definition:

- ***Property 1***: Every happy person drives at least one vehicle.
- ***Property 2***: All vehicles are driven by happy people.
- ***Property 3***: Happy persons only drive vehicles/cars they like.

Note that while Properties 1 and 2 are formulated entirely in the language of the target metamodel, Property 3 involves both metamodels and assumes backward traceability of happy persons and vehicles to persons and cars, respectively. We expect Property 1 and Property 3 to hold true for the MT definition. For Property 2, we expect to find a counterexample, since according to the MT definition in Sect. 4.1.3, all cars should be translated to vehicles – never mind they are not owned or liked by somebody. Thus, some of these vehicles should not be driven by any happy person.

```
//--------Property definition-----------
//Prop1 : Every happy person drives at least one vehicle.
assert ass1 {
  all p:HappyPerson | some v:Vehicle | p->v in drives
}
//Prop2: all vehicles are driven by happy people.
assert ass2 {
  all v:Vehicle | some h:HappyPerson | h->v in drives
}
//Prop3: Happy persons only drive vehicles/cars they like.
assert ass3 {
  all h:HappyPerson | all  v:Vehicle |
      h->v in drives => map1[h]->map2[v] in likes
}
```

Figure 6.5: Properties specified in Alloy

Having the Alloy specification presented in Fig. 6.2, we can formally define the *Properties 1-3* in Alloy as they are listed in Fig. 6.5. These properties are defined inside assertion blocks that let us use the Alloy analyzer to check their validity. As mentioned in Sect. 6.1.4, the Alloy analyzer works based on scope parameters; that is, assertions are checked within a user-defined scope that limits the maximum number of elements of each set. In the case that Alloy finds a counterexample within the defined scope, it means the assertion is not valid; otherwise, the assertion is valid within the scope and *might* be valid beyond it. We checked the three properties with the number 20 assigned to the scope parameters of the sets. The analyzer did find a counterexample for the second assertion refuting its correctness: there might be some vehicles without a driver in the generated models. Fig. 6.6 exhibits one of these counterexamples generated by the Alloy analyzer. In the figure, the source model elements are presented in gray, the generated target model elements are presented in blue, the query results are presented in green, and the traceability links are presented in red (labeled with *map1* and *map2*). *Vehicle1* from the target model is an example of a vehicle that is not driven by anybody as indicated in the figure.

For the first and the third property (i.e., the assertions `ass1` and `ass2` in Fig. 6.6), Alloy could not find any counterexamples, meaning that they *might* be valid even for an unlimited scope. One way to increase certainty about their validity is to increase the scope parameter value to a higher number to cover more model instances; but no matter how much we increase the scope boundaries, the uncertainty will never disappear. To gain greater confidence, we need to provide a formal proof (which

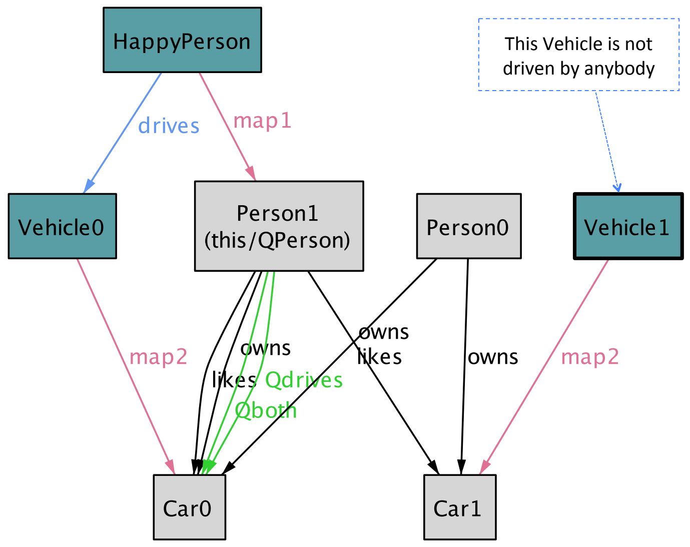


Figure 6.6: A counterexample generated by the Alloy analyzer for Prop. 2

gives us absolute confidence modulo the adequacy of the formal encoding of the subject matter). This could be done manually, or semi-automatically with a theorem prover; however, as mentioned before, provability of FOL statements is undecidable in general; the latter, of course, does not prevent the existence of decision procedures for specific fragments and cases.

In the following, we will manually prove both of the above two properties.

Let $\boldsymbol{Th}_H$ denote the encoded theory of the HappyPeople MT as specified in Fig. 6.2.

**Theorem 1** (Property 1 Holds). *Every happy person drives at least one vehicle, formally:* $\boldsymbol{Th}_H \models \forall h{:}\texttt{HappyPerson}, \exists v{:}\texttt{Vehicle}, v \in h.\texttt{drives}$.

*Proof.* In fact, we need to prove that the multiplicity of relation `drives` is [1..*]. However, relations `drives` and `Qdrives` are isomorphic by the construction of $\boldsymbol{Th}_H$ (lines 16–21 in Fig. 6.2), and so by Fact 2 (see Sect. 6.1.5) we need to prove the [1..*]-multiplicity of the relation `Qdrives`. As the relation `Qdrives` is total by construction (line 30 in Fig. 6.2), relation `Qdrives` does have multiplicity [1..*]. □

**Theorem 2** (Property 3 Holds). *If a happy person drives a vehicle, this person likes the corresponding car in the source model. Formally,* $\boldsymbol{Th}_H \models$
$\forall h{:}\texttt{HappyPerson}, \forall v{:}\texttt{Vehicle},\ v \in h.\texttt{drives} \ \Rightarrow\ map2(v) \in map1(h).\texttt{likes}$

*Proof.* As relations `drives` and `Qdrives` are isomorphic, we need to prove the property in question for relation `Qdrives`: $\boldsymbol{Th}_H \models c \in p.\texttt{Qdrives} \ \Rightarrow\ c \in p.\texttt{likes}$ for all $p \in \texttt{QPerson}$, $c \in \texttt{Car}$ (recall that `QPerson` is a subset of `Person`). Now it is easy to see that the property above means that $\texttt{Qdrives} \subseteq \texttt{likes}$, which is obvious as $\texttt{Qdrives} = \texttt{Qboth}$ and $\texttt{Qboth} = \mathit{owns} \cap \texttt{likes}$; $so, \texttt{Qdrives} \subseteq \texttt{likes}$. □

**Remark 2.** *When we need to prove a property that involves target metamodel elements (which is a typical case), we replace those elements by their images in $Q(\mathbb{S})$ and rewrite the property accordingly.*

### 6.4.2 Analyzing the ClassToTable Example

The property analysis task in the ClassToTable example is the same as the HappyPeople example. In fact, the more complex an MT gets, the more difficult the proofs might be; however, using the Alloy analyzer, an analysis task such as refuting correctness of a property might be very easy even for a fairly complex MT such as the ClassToTable example[1], as we will see in this section. We intend to check the following three properties over the ClassToTable example.

- ***Property 1***: The number of generated tables is equal to the number of classes.
- ***Property 2***: All the generated primary keys for a table are also columns of the same table.
- ***Property 3***: All the generated tables have at least one foreign key.

In the first property, both of the source and target metamodels are involved. In the second and the third property, the properties are regarding the validity of the generated models, so only the target metamodel is involved. Since we expect that in some ClassToTable transformation scenarios, the number of generated tables become greater than the number of classes in the source, if the ClassToTable MT definition was correct, Property 1 should not always hold true. There is also no restriction on the generated tables as to always have foreign keys (according to the ClassToTable MT specification –see Sect. 5.2.2), so we expect Property 2 to be also refuted. However, we expect the MT definition to have the second property; otherwise, the generated target models will not be valid database schemas. Note that the user can define many of other properties considering different aspect of the ClassToTable transformation specification and analyze the MT definition from multiple perspectives. The above three properties are just examples to show the applicability of the approach.

Given the encoding of the ClassToTable example in Appendix A.1 (that is partially explained in Sect. 6.3), we can express the above three properties in Alloy using the assertion blocks. Fig. 6.7 presents these assertion definitions. The sharp (i.e., #) notation at lines 3 and 13 is the cardinality operator. In line 8, the `in` keyword (i.e.,

[1]The ClassToTable specification is more than 200 lines of code.

```
1  // ----Property 1----
2  assert ass1 {
3    #QTable = #Class
4  }
5
6  // ----Property 2----
7  assert ass2{
8    QpKeys in Qcols
9  }
10
11 // ----Property 3----
12 assert ass3 {
13   all t:QTable | #(t.QfKeys) >=1
14 }
```

Figure 6.7: ClassToTable sample properties specified in Alloy

the subsetting relation) is used over two relations *QpKeys* and *Qcols* as sets of pairs. The other parts of the encodings are self-descriptive. As seen in these assertions, since we did not encod the target metamodel in the ClassToTable example, target types are substituted by their corresponding query definitions in the proposition encodings; for example, in the assertion at line 3, instead of `Table`, we used its corresponding query `QTable`.

We checked the assertions in Fig. 6.7 over the ClassToTable MT specification with the number 20 assigned to the scope parameters of the sets. The Alloy analyzer did find counterexamples for *Property 1* and *Property 3*, refuting their correctness. For the first property, Alloy provided the counterexample presented in Fig. 6.8. The property asserts that the number of generated tables should be equal to the number of classes. As seen, there exists only one class in the source model, while two tables are generated in the target. The extra table (labeled as `QTable1`) is generated because of one multi-valued attribute labeled as `Attribute` in the figure. Note how the generated traceability link `r3` traces back the generated table (i.e., `QTable1`) to its origin (i.e., `Attribute`) providing useful information regarding the invalidity of the property.

A counterexample generated for *Property 3* is presented in Fig. 6.9. The property asserts that all the generated tables should have at least one foreign key. As seen in the figure, there exists no foreign key associated with the table labeled as `QTable1`. Note again the traceability link `r2` explicitly tracing this table back to its origin (i.e., `Class`) making it possible to spot the place in the source model that causes the violation of the stated property.

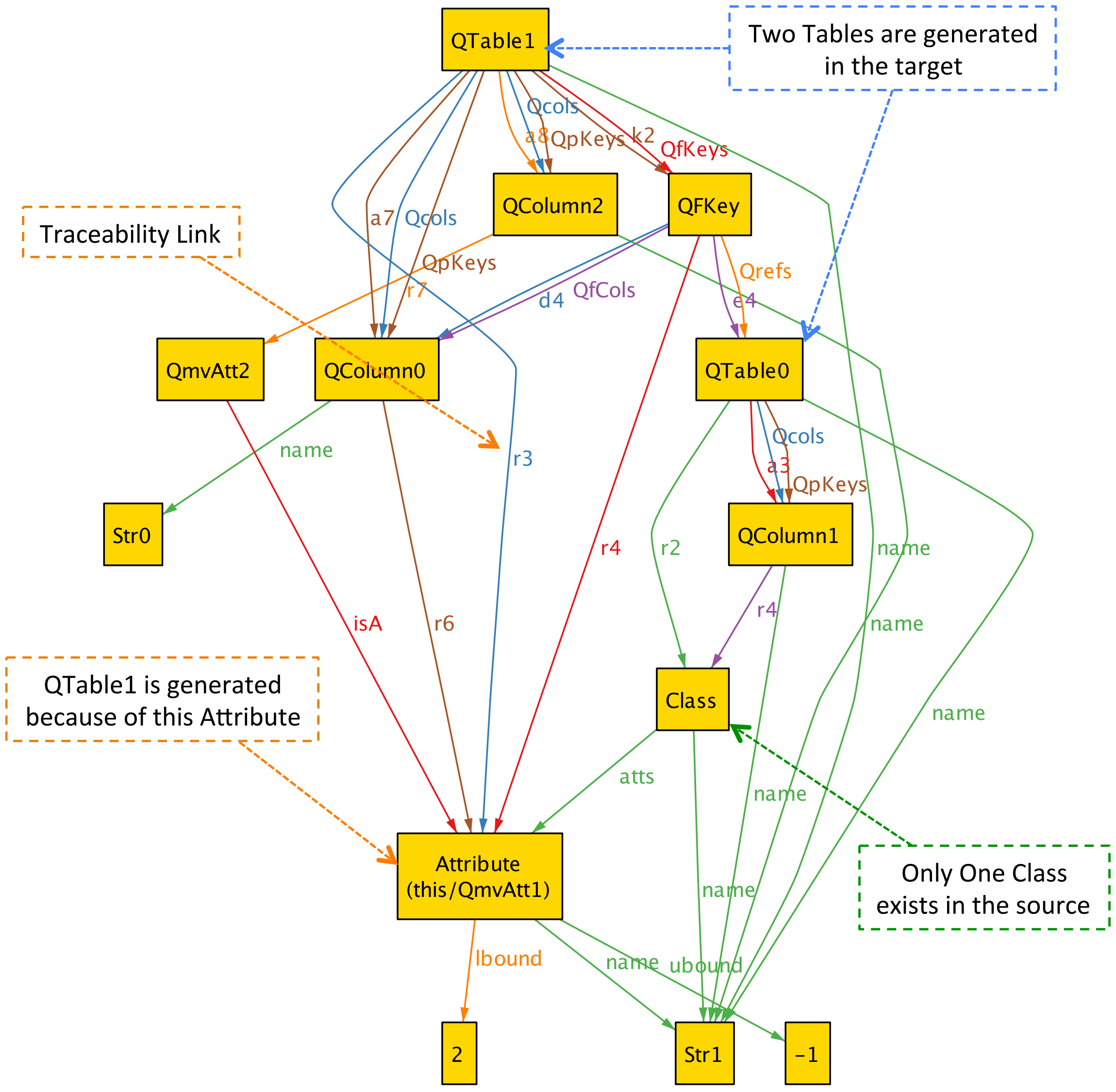


Figure 6.8: A counterexample provided by the Alloy analyzer for *Property 1*.

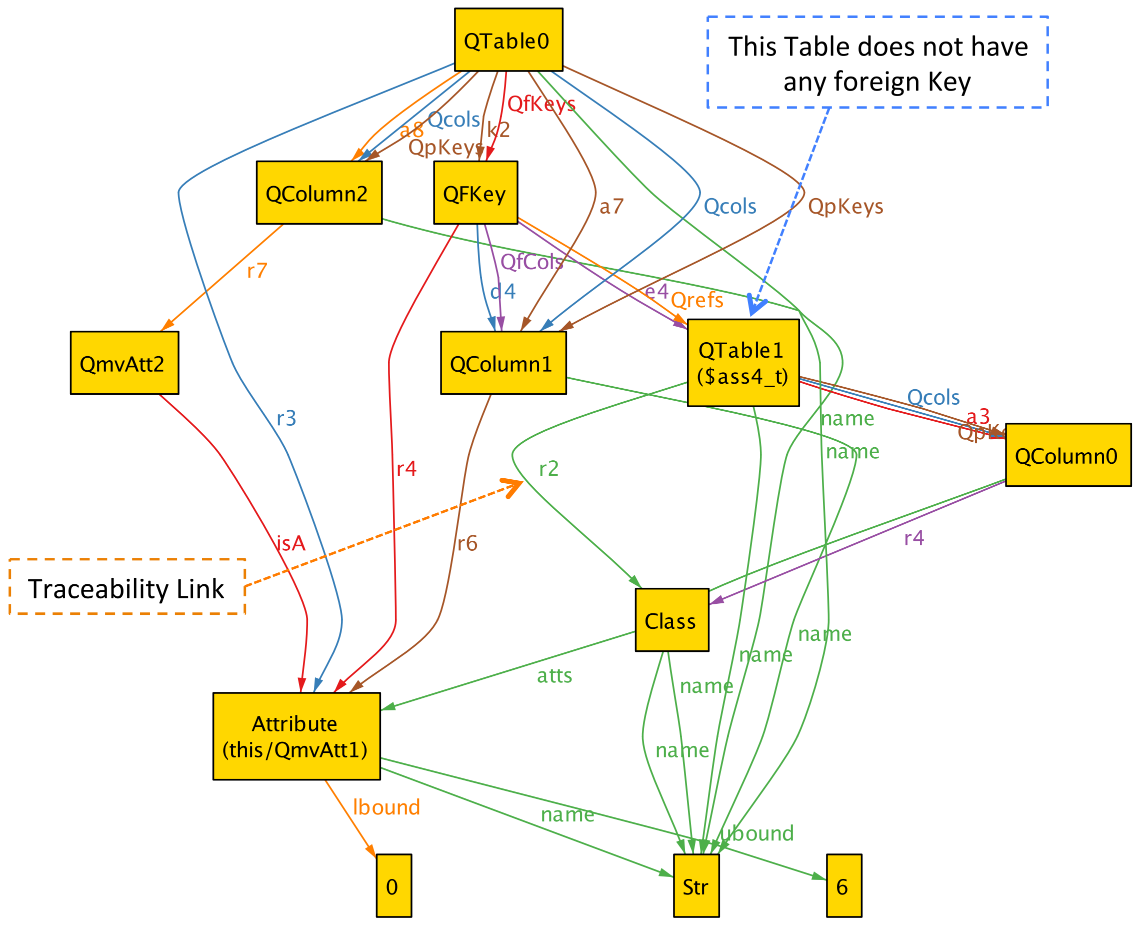


Figure 6.9: A counterexample provided by the Alloy analyzer for *Property 3*.

The Alloy analyzer could not find any counterexample for *Property 2* at the defined scope boundaries (i.e., 20 elements per sets). Increasing the scope boundaries to 30 also did not return any counterexample, making the property more likely to be a theorem; however, one might want to prove the property. Let $\boldsymbol{Th}_C$ denote the theory of the ClassToTable example as specified in Appendix A.1.

**Theorem 3** (Property 2 Holds)**.** *All the generated primary keys for a table are also columns of the same table:* $\boldsymbol{Th}_C \models QpKeys \subseteq Qcols$.

*Proof.* Straightforward from the `QpKeys` definition at line 147 and the `Qcols` definition at line 127 of the $\boldsymbol{Th}_C$ specification in Appendix A.1. □

## 6.5 Conclusion

There exist two general approaches to checking the validity of the properties of an MT definition: 1) testing 2) mathematical analysis (i.e., formal methods). The first approach cannot guarantee the correctness of a property of a transformation, except for the test models over which the property is being checked. The second approach can provide a higher level of confidence by proving the correctness of a property for all the possible inputs. The current MT approaches do not often provide a mathematical basis to formally analyze MT properties, except the graph-based approaches (see Sect. 1.3 for further discussion). In this chapter, we explained the techniques based on which properties over QueST MT definitions can be formally specified and verified.

Each QueST MT definition amounts to its corresponding theory and checking properties amounts to proving/disproving the correctness of the propositions (corresponding to the properties) in this theory. We elucidated the way this mechanism works using two examples (i.e., ClassToTable and HappyPeople). We presented the theories underlying these two MT definitions by encoding them in Alloy. After we carried out the theory encodings, we defined some properties and expressed them as Alloy assertions/propositions. We used the Alloy analyzer to verify these properties. The analyzer disproved some of the properties; however, it could not find counterexamples to refute the correctness of some others. The non-refuted properties, then, are highly likely to be correct. We manually proved these properties to be theorems in the corresponding MT theories.

The logical analysis technique of QueST is independent of Alloy and its analyzer tool. Whether Alloy is much suitable for QueST MT analysis than some other formal languages could be another research work. As we did not intent to compare the formal languages and their corresponding tools in this chapter, we did not provide performance data concerning the executed MT analyses with Alloy. The encoding process of a QueST MT presented in this chapter is carried out manually. The metamodels are encoded as a family of sets and relations, and the view-mappings are encoded as bijections. Queries are encoded via adding metamodel elements and constraints that semantically bound these new elements to the query inputs (according to their corresponding semantics). The definition of each QueST query has a predefined syntax (see Definition 10); it syntactically augments the metamodel in a predefined manner (see Definition 12); it also should have given a predefined semantics –independently of any specific MT definition (see Definition 14). Thus, it is possible to systematically automate the Alloy encoding of each QueST query application. We left the further investigation and the implementation of automatic encoding for future work. The presented analysis methods in this chapter can be employed over any QueST MT definition.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

The current thesis can be seen as a bridge between an engineering domain (MDE) and its abstract mathematical model (QueST). It attempts to make the two sides closer, and communicate QueST's promising advantages to the MDE community.

MDE proposes engineering software systems based on models rather than code. Engineers build models of software, and these models are transformed to executable code in a series of transformations. However, this promising idea is difficult to achieve in practice. Models are complex entities and each might have thousands of elements; engineering transformations between such complex entities and maintaining them is a challenging task. Further, checking each transformation's correctness is crucial to ensuring the integrity of the entire network of interrelated models. All this places the MT approaches at the heart of the MDE processes and emphasizes the critical role of the declarative MT approaches to the success of MDE.

Concretely, this thesis contributes in the following aspects: 1) studying the various types of synchronization scenarios, 2) formal definition of the structure of diagrammatic queries, 3) comparative analysis of the QueST approach and the current rule-based approaches, 4) applicability of the QueST approach in the development of modularity, incrementality, and logical analysis of MT definitions.

Below, we briefly summarize the findings.

**Shortcomings of the Current MT Approaches.** Examining the current MT approaches in the literature reveals that they usually miss either one or both of the two necessary features in managing the complexity of MT definitions: 1) declarativity, 2) formal foundation. The declarativity abstracts away unnecessary implementation details and focuses the user on high-level structures (i.e., models). In fact, using abstraction is more or less what MDE is about, so declarative approaches should be firmly expected to be used in the development of MT definitions. The formality, on the other hand, is essential to MT tool development, MT engineering, and MT verification.

**Query-based Structured MT Approach.** In the current thesis, we developed a Query-based Structured Model Transformation (QueST) approach that is formal and more declarative than the current MT approaches. The formality behind the QueST approach has been already well-established in the literature based on Category Theory concepts. In the current thesis, we further developed the approach by introducing a diagrammatic query framework (DQF). The declarativity of the approach is achieved by defining the MTs at the metamodel level and hiding the control flow structures under the declarative query definition constructs.

**MT Definition Process in QueST.** The main components of an MT definition in QueST are a series of queries defined on source metamodels. Discovering these definitions is a heuristic process, guided by the structure of the corresponding target metamodel; the goal is to replicate the target metamodel structure on the source side by defining queries. After that, the elements of the target metamodel are mapped to either types defined by the queries or the original source types. This is similar to how views are defined in relational databases; however, in QueST, views are defined over metamodels rather than relational schemas.

**High-level Diagrammatic Queries.** The queries used in defining transformations in QueST are *high-level diagrammatic* operations applied to the metamodel elements. They are called diagrammatic as their corresponding input/output arities are graphs. They are considered high-level as their corresponding input/output parameters are collections rather than individual instances. We introduced the diagrammatic query framework (DQF) based on which the syntax and the semantics of these queries are defined. We provided formal definitions of the framework components, and explained them with concrete examples. We also showed that the query definitions in the framework are aligned with the general concept of queries in terms of leaving the original data intact.

**Comparative Analysis.** We compared QueST and three rule-based approaches: ETL, ATL, and QVT-R. QueST guides the MT designer's thinking process in the *target-to-source* direction, and focuses it at the *metamodel* level, while the rule-based approaches guide it in the *source-to-target* direction and focus it at the *model* level. QueST queries are more declarative constructs than are rules. They process models as collections and abstract away the implementation details (i.e., control flow structures). In contrast, rules process models element-by-element, and the control flow structures sometimes appear either inside the rules or as triggering dependencies between the rules. We argued that the decomposition of MT definitions in QueST is more flexible than rule-based approaches. The reason is that the QueST queries can be chained and construct MT definitions incrementally, while the structures of rules are not amenable for building MT definitions incrementally.

**MT Verification in QueST.** MT verification is critical to ensuring the correctness of the transformations. The current MT approaches typically do not provide a mathematical basis to formally analyze MT properties. We explained the technique based on which properties over QueST MT definitions can be formally specified and verified. Each QueST MT definition amounts to its corresponding theory, and checking transformation properties amounts to proving/disproving the derivability of the propositions. We elucidated the way this mechanism works for QueST MT definitions. We have also demonstrated how these properties can be checked/analyzed with Alloy.

## 7.2 Future Work

**DQF Query Language.** Ahough we defined the DQF framework, we did not confine it to any particular set of diagrammatic query operations. One can define a set of DQF operations constructing a specific DQF query language and investigate its properties such as expressivity (e.g., whether the language is turing complete).

**Language-level DQF Query Compositions.** We showed the way query definitions can be composed consecutively in a QueST MT definition; however, we did not define the machinery of composing two DQF operations in a DQF language, independently of their definitions on a metamodel. This composition mechanism and the corresponding semantic definitions might be further investigated.

**Alloy Automatic Encoding.** We explained the encoding techniques, but have not provided a tool to automatically encode QueST MTs as Alloy specifications. An automatic encoding mechanism might be implemented as part of a DQF implementation. A tentative approach would be as follows: each diagrammatic query operation in a DQF library comes with its Alloy encoding template. Whenever a query is used in a QueST MT definition, the encoder uses its corresponding template to generate the encoding in Alloy.

**Empirical Studies.** We compared QueST with three rule-based approaches. There might be a possibility to carry out some empirical studies over the usability of QueST in contrast to these approaches. However, this requires providing tool support for the QueST approach comparable to that available for the rule based approaches.

# Bibliography

Ab. Rahim, L. and Whittle, J. (2015). A survey of approaches for verifying model transformations. *Softw. Syst. Model.*, **14**(2), 1003–1028.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, **1**(1), 68–88.

Amrani, M., Combemale, B., Lúcio, L., Selim, G., Dingel, J., Le Traon, Y., Vangheluwe, H., and Cordy, J. R. (2014). Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, pages 1–43.

Anastasakis, K., Bordbar, B., and Küster, J. M. (2007). Analysis of model transformations via Alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56.

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer.

Arlow, J. and Neustadt, I. (2005). *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Addison-Wesley Professional.

Balasubramanian, D., Narayanan, A., van Buskirk, C., and Karsai, G. (2007). The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST*, **1**.

Barroca, B., Lúcio, L., Amaral, V., Félix, R., and Sousa, V. (2011). *DSLTrans: A Turing Incomplete Transformation Language*, pages 296–305. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., and Ökrös, A. (2010). Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems*, pages 76–90. Springer.

Bertot, Y. and Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Beydeda, S., Book, M., Gruhn, V., *et al.* (2005). *Model-driven software development*, volume 15. Springer.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J., and Stefanescu, M. (2002). XQuery 1.0: An XML query language.

Buneman, P. (1997). Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121. ACM.

Buneman, P., Fernandez, M., and Suciu, D. (2000). UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal—The International Journal on Very Large Data Bases*, **9**(1), 76–110.

Büttner, F., Egea, M., Cabot, J., and Gogolla, M. (2012). Verification of ATL transformations using transformation models and model finders. In *Formal Methods and Software Engineering*, pages 198–213. Springer.

Calegari, D., Luna, C., Szasz, N., and Tasistro, Á. (2011). A type-theoretic framework for certified model transformations. In *Formal Methods: Foundations and Applications*, pages 112–127. Springer.

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2010). JTL: a bidirectional and change propagating transformation language. In *Software Language Engineering*, pages 183–202. Springer.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., and Quesada, J. F. (2002). Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, **285**(2), 187–243.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, **13**(6), 377–387.

Cunha, A., Garis, A., and Riesco, D. (2013). Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, **14**(1), 5–25.

Date, C. J. and Darwen, H. (1997). *A Guide To SQL Standard*, volume 3. Addison-Wesley Reading.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1999). A query language for XML. *Computer networks*, **31**(11), 1155–1169.

Diskin, Z. (1996). Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems, Riga, Latvia.

Diskin, Z. (1997). Towards algebraic graph-based model theory for computer science. *Bulletin of Symbolic Logic*, **3**, 144–145.

Diskin, Z. (2008). Model transformation as view computation. *Technical Report CSRG-582 Department of Computer Science, University of Toronto.*

Diskin, Z. (2011). Model synchronization: Mappings, tiles, and categories. *Generative and Transformational Techniques in Software Engineering III*, pages 92–165.

Diskin, Z. and Dingel, J. (2006). A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In *ATEM*.

Diskin, Z. and Maibaum, T. S. E. (2012). Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond. In U. Golas and T. Soboll, editors, *ACCAT*, volume 93 of *EPTCS*, pages 1–21.

Diskin, Z. and Wolter, U. (2008). A Diagrammatic Logic for Object-Oriented Visual Modeling. *Electr. Notes Theor. Comput. Sci.*, **203**(6), 19–41.

Diskin, Z., Xiong, Y., and Czarnecki, K. (2011a). From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, **10**, 6: 1–25.

Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., and Orejas, F. (2011b). From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In Whittle *et al.* (2011), pages 304–318.

Diskin, Z., Maibaum, T., and Czarnecki, K. (2012). Intermodeling, Queries, and Kleisli Categories. In J. de Lara and A. Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 163–177. Springer.

Diskin, Z., Wider, A., Gholizadeh, H., and Czarnecki, K. (2014). Towards a rational taxonomy for increasingly symmetric model synchronization. In *International Conference on Theory and Practice of Model Transformations*, pages 57–73. Springer.

Diskin, Z., Gholizadeh, H., Wider, A., and Czarnecki, K. (2016). A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software*, **111**, 298–322.

Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Embley, D. W. and Thalheim, B. (2012). *Handbook of conceptual modeling: theory, practice, and research challenges*. Springer.

Fischer, T., Niere, J., Torunski, L., and Zündorf, A. (2000). Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In *Theory and Application of Graph Transformations*, pages 296–309. Springer.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society.

Freund, M. and Braune, A. (2016). A generic transformation algorithm to simplify the development of mapping models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 284–294. ACM.

Fuhr, Norbert and Großjohann, Kai (2001). XIRQL: A query language for information retrieval in XML documents. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 172–180. ACM.

Gammaitoni, L. and Kelsen, P. (2015). F-Alloy: An Alloy Based Model Transformation Language. In D. Kolovos and M. Wimmer, editors, *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, pages 166–180. Springer International Publishing.

Gholizadeh, H. (2013). Towards declarative and incremental model transformation. In *Proceedings of the MODELS 2013 Doctoral Symposium co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, October 1, 2013.*, pages 32–39.

Gholizadeh, H. (2014). QueST: Well-formed structure for model transformation. NECSIS Workshop, Kitchener-Waterloo, CA. Presentation, Unpublished.

Gholizadeh, H. (2015). A prototype implementation of the QueST framework in Alloy. NECSIS Workshop, Vancouver, CA. Presentation, Unpublished.

Gholizadeh, H., Diskin, Z., and Maibaum, T. (2014). A query structured approach for model transformation. In *Proceedings of the Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 29, 2014.*, pages 54–63.

Gholizadeh, H., Diskin, Z., Kokaly, S., and Maibaum, T. (2015). Analysis of source-to-target model transformations in QueST. In *Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 28, 2015.*, pages 46–55.

Goldman, R., McHugh, J., and Widom, J. (1999). From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *ACM SIGMOD Workshop on The Web and Databases, WebDB 1999, Philadelphia, Pennsylvania, USA, June 3-4, 1999. Informal Proceedings*, pages 25–30.

Guerra, E. and de Lara, J. (2012). An Algebraic Semantics for QVT-Relations Check-only Transformations. *Fundam. Inform.*, **114**(1), 73–101.

Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., and Xiong, Y. (2011). Correctness of Model Synchronization Based on Triple Graph Grammars. In Whittle *et al.* (2011), pages 668–682.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K. (2011). GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011), Oread, Lawrence, Kansas, USA*, pages 480–483. IEEE.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **11**(2), 256–290.

Jackson, D. (2012). *Software Abstractions: logic, language, and analysis.* MIT press.

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

Kalnins, A., Barzdins, J., and Celms, E. (2005). Model transformation language MOLA. In *Model Driven Architecture*, pages 62–76. Springer.

Kaur, K. and Rani, R. (2013). Modeling and querying data in NoSQL databases. In *Big Data, 2013 IEEE International Conference on*, pages 1–7. IEEE.

Kindler, E. and Wagner, R. (2007). Triple Graph Grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Technical Report tr-ri-07-284, University of Paderborn.

Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the Model Driven Architecture: practice and promise*. Addison-Wesley Professional.

Klyne, G. and Carroll, J. J. (2004). Resource Description Framework (RDF): Concepts and abstract syntax. W3C recommendation, 2004. *World Wide Web Consortium*.

Kolovos, D., Rose, L., Paige, R., and Garcıa-Domınguez, A. (2010). *The Epsilon Book*, volume 178. York University, http://www.eclipse.org/epsilon/doc/book/.

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006). The Epsilon Object Language (eol). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). *The Epsilon Transformation Language*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg.

Macedo, N. and Cunha, A. (2013). Implementing QVT-R bidirectional model transformations using Alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer.

Maoz, S., Ringert, J. O., and Rumpe, B. (2011). CD2Alloy: Class Diagrams analysis using Alloy revisited. In *Model Driven Engineering Languages and Systems*, pages 592–607. Springer.

Miller, J., Mukerji, J., *et al.* (2003). MDA guide version 1.0. 1. *Object Management Group*, **234**, 51.

OMG (February 2015). *Meta Object Facility (MOF) 2.0 query/view/transformation specification, version 1.2.* http://www.omg.org/spec/QVT/1.2.

Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and Complexity of SPARQL. In *International semantic web conference*, volume 4273, pages 30–43. Springer.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2nd edition.

Romero, J. R., Rivera, J. E., Durán, F., and Vallecillo, A. (2007). Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, **6**(9), 187–207.

Rutle, A. (2010). *Diagram Predicate Framework: A Formal Approach to MDE.* Ph.D. thesis, The University of Bergen, Norway.

Schmidt, D. C. (2006). Model-Driven Engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, **39**(2), 25.

Schürr, A. and Klar, F. (2008). 15 years of Triple Graph Grammars. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Graph Transformations: 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008.*, volume 5214, pages 411–425, Berlin, Heidelberg. Springer Berlin Heidelberg.

Selic, B. (2003). The pragmatics of Model-Driven Development. *IEEE Software*, **20**(5), 19–25.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of Model-Driven software development. *IEEE software*, **20**(5), 42–45.

Spivey, J. M. and Abrial, J. (1992). *The Z notation.* Prentice Hall Hemel Hempstead.

Stahl, T. and Völter, M. (2006). *Model-Driven Software Development.* J. Wiley & Sons.

Stahl, T., Völter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework.* Pearson Education.

Stevens, P. (2010). Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, **9**(1), 7–20.

Tarski, A. (1941). On the calculus of relations. *The Journal of Symbolic Logic*, **6**(03), 73–89.

Taylor, R. W. and Frank, R. L. (1976). CODASYL data-base management systems. *ACM Computing Surveys (CSUR)*, **8**(1), 67–103.

Troya, J. and Vallecillo, A. (2011). A rewriting logic semantics for ATL. *Journal of Object Technology*, **10**(5), 1–29.

Tsichritzis, D. and Lochovsky, F. H. (1976). Hierarchical data-base management: A survey. *ACM Computing Surveys (CSUR)*, **8**(1), 105–123.

Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, **98**, 80–99.

Warmer, J. and Kleppe, A. (2000). *The Object Constraint Language. Precise modeling with UML*. Addison-Wesley.

Webber, J. (2012). A programmatic introduction to Neo4J. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA. ACM.

Whittle, J., Clark, T., and Kühne, T., editors (2011). *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, volume 6981 of *Lecture Notes in Computer Science*. Springer.

Zhang, J., Lin, Y., and Gray, J. (2005). Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*, pages 199–217. Springer.

# Appendix A

# The ClassToTable Definitions

## A.1 The ClassToTable Example Encoding in Alloy (via QueST)

In Chapter 6, we presented some parts of the ClassToTable QueST encoding in Alloy. The complete encoding of this example is provided in the following.

```
open util/relation
//*************************************************
//-------------------Source metamodel-------------
//*************************************************
// ---metamodel nodes and arrows---
sig Str{}
abstract sig NElement {name: one Str}
sig Class extends NElement
 {parent :lone Class, atts: set Attribute}
sig Association extends NElement
 {lbound, ubound: one Int, src, trg: one Class}
sig Attribute extends NElement
 {lbound,ubound:one Int}

// ---constraints------------------
fact diamond{
 all a:Attribute | one c:Class | a in c.atts
}
fact noLoop{
 all c:Class | not (c in c.^parent)
}
fact bounds{
 all a:Association | a.lbound >=0
 all a:Association | not (a.lbound =0 and a.ubound =0)
 all a:Association | (a.lbound <= a.ubound) or (a.ubound=-1)

 all a:Attribute | a.lbound >=0
 all a:Attribute | not (a.lbound =0 and a.ubound =0)
 all a:Attribute | (a.lbound <= a.ubound) or (a.ubound=-1)
}
//****************************************************
//------------------------Queries---------------------
//****************************************************
// ---Helper queries to build QTable
// --QmvAtt1 definition-------------
//***************************
sig QmvAtt1 in Attribute{}
fact {
 all a:Attribute |
  (a.ubound =-1 or a.ubound >1) iff a in QmvAtt1
}
// --QmvAssoci1 definition-----
//***************************
sig QmvAssoci1 in Association{}
fact {
 all a:Association |
  (a.ubound =-1 or a.ubound >1) iff a in QmvAssoci1
}

//***************************
// -- QTable query:
// --        the Table node in target metamodel is mapped to QTable
//***************************
sig QTable extends NElement {
  r1: set QmvAssoci1,
   r2: set Class,
    r3: set QmvAtt1,
     a1,a2,a3,a4,a5,a6,a7,a8,Qcols, QpKeys: set QColumn,
      k1,k2,k3,k4,k5,QfKeys: set QFKey

}
fact {
 bijection[
   QTable<:r3+QTable<:r1+QTable<:r2,
    QTable,
     QmvAtt1+QmvAssoci1+Class]
}
// -- this is to populate the name property of QTable elements.
fact {
 all q:QTable | q.r1!=none  => q.name=q.r1.name else
   q.r2!=none => q.name=q.r2.name else
    q.r3!=none => q.name=q.r3.name
```

```
}

// ---Helper queries to build QColumn
// --QmvAssoci2 definition------------
//****************************
sig QmvAssoci2  { isA:QmvAssoci1 }
fact  {
 bijection[QmvAssoci2<:isA,QmvAssoci2, QmvAssoci1]
}
// --QmvAtt2 definition------------
//****************************
sig QmvAtt2 { isA:QmvAtt1 }
fact  {
 bijection[QmvAtt2<:isA,QmvAtt2, QmvAtt1]
}
// --QsvAssoci1 definition------------
//****************************
sig QsvAssoci1 in Association{}
fact {
 all a:Association |
  (a.ubound =1 and (a.lbound =1 or a.lbound =0))
   iff a in QsvAssoci1
}
// --QsvAtt1 definition------------
//****************************
sig QsvAtt1 in Attribute{}
fact {
 all a:Attribute |
  (a.ubound =1 and (a.lbound =1 or a.lbound =0))
   iff a in QsvAtt1
}
// --Qparent1 definition------------
//****************************
sig Qparent1 {cc : Class ->Class, s,t:Class}
 { one cc and t=cc[s]}
fact{
 parent=Qparent1.cc and #Qparent1=#parent
}
//****************************
// -- QColumn query:
// --        the Column node in target metamodel is mapped to QColumn
//****************************
sig QColumn extends NElement
 {r1: set QsvAssoci1, r2: set QmvAssoci1, r3: set QmvAssoci2,
   r4: set Class, r5: set Qparent1, r6: set QmvAtt1,
   r7: set QmvAtt2, r8: set QsvAtt1}
fact {
 bijection[QColumn<:r1+QColumn<:r2+QColumn<:r3+
        QColumn<:r4+QColumn<:r5+QColumn<:r6+
         QColumn<:r7+QColumn<:r8 ,
          QColumn,
           QsvAssoci1+QmvAssoci1+QmvAssoci2+Class+
            Qparent1+QmvAtt1+QmvAtt2+QsvAtt1]
}

//****************************
// -- Qcols query:
// --        the cols arrow in target metamodel is mapped to Qcols
//****************************
fact {
 a1 = (QTable<:r1).~r2
 a2 = (QTable<:r1).~(QmvAssoci2<:isA).~r3
 a3 = (QTable<:r2).~r4
 a4 = (QTable<:r2).atts.~r8
 a5 = (QTable<:r2).~(Qparent1<:s).~r5
 a6 = (QTable<:r2).~src.~r1
 a7 = (QTable<:r3).~r6
 a8 = (QTable<:r3).~(QmvAtt2<:isA).~r7
 Qcols=a1+a2+a3+a4+a5+a6+a7+a8
}

//****************************
// -- QFKey query:
// --        the FKey node in target metamodel is mapped to QFKey
//****************************
sig QFKey {r1: set QmvAssoci1, r2: set QmvAssoci2,
     r3: set Qparent1, r4: set QmvAtt1,r5: set QsvAssoci1,
      e1,e2,e3,e4,e5, Qrefs: set QTable,
       d1,d2,d3,d4,d5,QfCols:set QColumn}
fact{
  bijection[QFKey<:r1+QFKey<:r2+QFKey<:r3+
      QFKey<:r4+QFKey<:r5,
       QFKey,
        QmvAssoci1+QmvAssoci2+Qparent1+
         QmvAtt1+QsvAssoci1]
}

//****************************
// -- QpKeys query:
// --        the pKeys arrow in target metamodel is mapped to QpKeys
//****************************
fact {
 QpKeys=a1+a2+a3+a7+a8
}
//****************************
// -- QfKeys query:
```

```
// --          the fKeys arrow in target metamodel is mapped to QfKeys
//***************************
fact {
 k1= (QTable<:r2).~(Qparent1<:s).~(QFKey<:r3)
 k2= (QTable<:r3).~(QFKey<:r4)
 k3= (QTable<:r1).~(QFKey<:r1)
 k4= (QTable<:r1).~(QmvAssoci2<:isA).~(QFKey<:r2)
 k5= (QTable<:r2).~src.~(QFKey<:r5)ctod-encoding

 QfKeys=k1+k2+k3+k4+k5
}
//***************************
// -- Qrefs query:
// --          the refs arrow in target metamodel is mapped to Qrefs
//***************************
fact {
 e1= (QFKey<:r1).src.~(QTable<:r2)
 e2= (QFKey<:r2).isA.trg.~(QTable<:r2)
 e3= (QFKey<:r3).t.~(QTable<:r2)
 e4= (QFKey<:r4).~atts.~(QTable<:r2)
 e5= (QFKey<:r5).trg.~(QTable<:r2)
 Qrefs=e1+e2+e3+e4+e5
}
//***************************
// -- QfCols query:
// --          the fCols arrow in target metamodel is mapped to QfCols
//***************************
fact {
 d1= (QFKey<:r1).~(QColumn<:r2)
 d2= (QFKey<:r2).~(QColumn<:r3)
 d3= (QFKey<:r3).~(QColumn<:r5)
 d4= (QFKey<:r4).~(QColumn<:r6)
 d5= (QFKey<:r5).~(QColumn<:r1)
 QfCols=d1+d2+d3+d4+d5
}
```

## A.2 The ClassToTable Example in ETL

In the following four pages, an ETL implementation of the ClassToTable example in Chapter 5 is provided.

```
pre {
  "Running␣ETL".println();
  var dbschema : myDB!DBSchema;
}
```

```

rule ClassSchema2DBSchema
  transform c : myOO!Schema
  to t:myDB!DBSchema{
   dbschema=t;
 }
rule Class2Table
  transform c : myOO!Class
   to t:myDB!Table,
    pk:myDB!Column{

   dbschema.tables.add(t);
   dbschema.tables.add(pk);
   t.name=c.name;
   pk.name=c.name+"PK";
   pk.type="INT";

   t.cols.add(pk);

   t.pKeys.add(pk);

   if (not (c.parent=null)){
    var col:new myDB!Column;
    col.name="parent-"+c.parent.name+"PK";
    col.type="INT";

    t.cols.add(col);

    var f:new myDB!Fkey;
    f.fCols.add(col);
    t.fKeys.add(f);
    f.ref=c.parent.getCorrTable();
   }

   for (a : myOO!Attribute in c.atts) {
  if (a.ubound = 1 ){

      t.cols.add(a.equivalent("svAtt2Col"));

  }
 }
}
rule mvAtt2Table
  transform a : myOO!Attribute
  to t:myDB!Table,
    c1:myDB!Column,
     c2:myDB!Column,
      f:myDB!Fkey{
```

```

   guard : (a.ubound> 1 or a.ubound=-1)

   dbschema.tables.add(t);
   t.name=a.name;
   c1.name=a.name;
   c1.type=a.type;

   var oclass:myOO!Class=a.getOwningClass();
   c2.name=oclass.name+"PK";
   c2.type="INT";

   t.cols.add(c1);
   t.cols.add(c2);

   t.fKeys.add(f);
   f.ref=oclass.getCorrTable();
   f.fCols.add(c2);
}
rule mvAssoci2Table
  transform a : myOO!Association
  to t:myDB!Table,
    c1:myDB!Column,
     c2:myDB!Column,
      f1:myDB!Fkey,
       f2:myDB!Fkey{

   guard : (a.ubound> 1 or a.ubound=-1)

   dbschema.tables.add(t);

   t.name=a.name;
   c1.name=a.src.name+"PK";
   c1.type="INT";
   c2.name=a.trg.name+"PK";
   c2.type="INT";
   f1.fCols.add(c1);
   f2.fCols.add(c2);
   f1.ref=a.src.getCorrTable();
   f2.ref=a.trg.getCorrTable();

   t.cols.add(c1);
   t.cols.add(c2);

   t.fKeys.add(f1);
   t.fKeys.add(f2);

}
```

```

rule svAtt2Col
  transform a : myOO!Attribute
  to c:myDB!Column{

   guard : (a.ubound = 1 )

   c.name=a.name;
   c.type=a.type;
}
rule svAssoci2Col
  transform a : myOO!Association
  to c:myDB!Column,
    f1:myDB!Fkey{

   guard : (a.ubound = 1 )

   c.name=a.name+"-"+a.trg.name+"PK";
   c.type="INT";

   a.src.getCorrTable().cols.add(c);

   f1.fCols.add(c);
   f1.ref=a.trg.getCorrTable();
   a.src.getCorrTable().fKeys.add(f1);
}
operation myOO!Attribute getOwningClass() : myOO!Class {
  return myOO!Class.all().selectOne(c|c.atts.includes(self));
}
operation myOO!Class getCorrTable(): myDB!Table {
 var tbs:Bag= self.equivalents("Class2Table").flatten();
   //tbs.println();
   var t:myDB!Table=null;
   for (k in tbs){
    if (k.isTypeOf(myDB!Table)){
     t=k;
    }
   }
    return t;
}
```

## A.3 The ClassToTable Example in ATL

In the following four pages, an ATL implementation of the ClassToTable example in Chapter 5 is provided.

```
-- @path myDB=/ATLClass2Table/metamodels/MyDBv3.ecore
-- @path myOO=/ATLClass2Table/metamodels/MyOOv3.ecore


module ClassToTable;
create UniversityDBSchema: myDB from UniversityClassDiagram: myOO;

entrypoint rule Metamodel() {
 do {
  'ATL Is running Class To Table Example'.println();
 }
}

rule Class2Table {
 from
  c: myOO!Class (
   c.parent.oclIsUndefined()
  )
 to
  t: myDB!Table (
   name <- c.name,
   pKeys <- Set{pk},
   cols <- c.svAtts -> union(Set{pk})
  ),
  pk: myDB!Column (
   name <- c.name + 'PK',
   type <- 'INT'
  )
}

rule SubClass2Table {
 from
  c: myOO!Class (
   not c.parent.oclIsUndefined()
  )
 to
  t: myDB!Table (
   name <- c.name,
   pKeys <- Set{pk},
   cols <- c.svAtts -> union(Set{pk}) -> union(Set{refCol}),
   fKeys <- fkey
  ),
  pk: myDB!Column (
   name <- c.name + 'PK',
   type <- 'INT'
  ),
  refCol: myDB!Column (
   name <- 'parent-' + c.parent.name + 'PK',
    type <- 'INT'
   ),
   fkey: myDB!Fkey (
    owner <- c,
    ref <- c.parent,
    fCols <- refCol
   )
}

rule svAtt2Col {
 from
  a: myOO!Attribute (
   a.ubound = 1
  )
 to
  c: myDB!Column (
   name <- a.name,
   type <- a.type
  )
}

rule mvAtt2Table {
 from
  a: myOO!Attribute (
   a.ubound > 1 or a.ubound = -1
  )
 to
  t: myDB!Table (
   name <- a.name,
   pKeys <- Set{c1,
      c2},
   cols <- Set{c1,
      c2}
  ),
  c1: myDB!Column (
   name <- a.name,
   type <- a.type,
   owner <- t
  ),
  c2: myDB!Column (
   name <- a.name + 'PK',
   type <- 'INT',
   owner <- t
  ),
  f1: myDB!Fkey (
   owner <- t,
   ref <- a.owner,
   fCols <- c2
  )
}

rule svAssoci2Col {
 from
  a: myOO!Association (
   a.ubound = 1
  )
 to
  c: myDB!Column (
   name <- a.name + a.trg.name,
   type <- 'INT',
   owner <- a.src
  ),
  f: myDB!Fkey (
   owner <- a.src,
   ref <- a.trg,
   fCols <- c
  )
}

rule mvAssoci2Table {
 from
  a: myOO!Association (
   a.ubound > 1 or a.ubound = -1
  )
 to
  t: myDB!Table (
   name <- a.name,
   pKeys <- Set{c1,
      c2},
   cols <- Set{c1,
      c2}
  ),
  c1: myDB!Column (
   name <- a.src.name + 'PK',
   type <- 'INT',
   owner <- t
  ),
  c2: myDB!Column (
   name <- a.trg.name + 'PK',
   type <- 'INT',
   owner <- t
  ),
  f1: myDB!Fkey (
   owner <- t,
   ref <- a.src,
   fCols <- c1
```

```
    ),
    f2: myDB!Fkey (
     owner <- t,
     ref <- a.trg,
     fCols <- c2
    )
}



rule ClassDiagram2DBSchema {
 from
  s: myOO!ClassDiagram
 to
  t: myDB!DBSchema (
   tables <- s.classes -> union(s.getAllmvAssoci()) -> union(s.getAllmvAtt())
  )
}

helper context myOO!ClassDiagram def: getAllmvAtt(): Set(myOO!Attribute) =
 myOO!Attribute.allInstances() -> select(a | a.ubound > 1 or a.ubound = -1);

helper context myOO!ClassDiagram def: getAllmvAssoci(): Set(myOO!Association) =
 myOO!Association.allInstances() -> select(a | a.ubound > 1 or a.ubound = -1);

helper context myOO!Class def: svAtts: Set(myOO!Attribute) =
  self.atts -> select(a | a.ubound = 1);
```

## A.4 The ClassToTable Example in QVT-R

In the following four pages, a QVT-R implementation of the ClassToTable example in Chapter 5 is provided.

```
--------------------------------------------------------------------------------
--- QVT-R transformation Definition for Class Diagram to DB Schema
--- Written by : Hamid Gholizadeh
--- version :v 1.0
--- http://www.golizadeh.net
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------

transformation c2t(OO:myOO, DB:myDB) {
    ---------------------------------------------------
 -- Relates each Class Diagram Schema to a schema
 ---------------------------------------------------
```

```
top relation ClassSchema2DBSchema {
  checkonly domain OO cs :myOO::Schema  {};
  enforce domain DB ds : myDB::DBSchema {};
}
---------------------------------
-- relating Classes To Tables
---------------------------------
top relation ClassToTable {
  cn : String;

  checkonly domain OO cl : myOO::Class {
    classSchema = cls:myOO::Schema{},
    name = cn
  };
  enforce domain DB tb : myDB::Table {
   schema = dbs:myDB::DBSchema{},
    name = cn
  };

  when {
   ClassSchema2DBSchema(cls, dbs);
  }
  where {
   ClassToCol(cl,tb); --requires class attributes also relate to table columns
  }
}
--------------------------------------
-- creating pk column for each class
--------------------------------------
relation ClassToCol {
  cn : String;
  checkonly domain OO cl : myOO::Class {
    name = cn
  };
  enforce domain DB tb : myDB::Table {
   cols=col1 : myDB::Column{name = cn+'_pk',type='INT'},
   pKeys=col1: myDB::Column{}
  };

}
---------------------------------------------------------
-- creating a column for every single value attribute
---------------------------------------------------------
top relation svAttsToCol {
   an : String;
   t:String;

  checkonly domain OO at : myOO::Attribute {
    name = an,
    owner=ow1:myOO::Class{},
    type=t
  };
  enforce domain DB tb : myDB::Column {
    name = an,
    owner=ow2:myDB::Table{},
    type=t
  };
  when {
   not (at.ubound > 1 or at.ubound=-1);
   ClassToTable(ow1,ow2);
  }
  where {
  }
}
-----------------------------------------------------------
-- creating a table for each multi-valued Attribute
-----------------------------------------------------------
top relation mvAttToTable {
  an : String;
  col1:myDB::Column;
  col2:myDB::Column;
  ownerTable:myDB::Table;

  checkonly domain OO at : myOO::Attribute {
    name = an,
    owner=ownerClass:myOO::Class{classSchema = cls:myOO::Schema{}}
  };
  enforce domain DB tb : myDB::Table {
    name = an,
    schema = dbs:myDB::DBSchema{}
  };
  when {
   ClassSchema2DBSchema(cls, dbs);
   (at.ubound > 1 or at.ubound=-1);
   ClassToTable(ownerClass,ownerTable);
  }
  where {
   mvAttsToCol(at,tb,col1,col2);
   mvAttsToFkey(at,tb,ownerTable);
  }
}
---------------------------------------------------------
-- creating two col for each multi-valued attribute
---------------------------------------------------------
relation mvAttsToCol {
   an : String;
   t:String;

  checkonly domain OO at : myOO::Attribute {
    name = an,
    type=t
  };

  enforce domain DB tb : myDB::Table {
  };
  enforce domain DB col1 : myDB::Column {
    name = an,
    owner=tb,
    type=t
  };
  enforce domain DB col2 : myDB::Column {
    name = at.owner.name+'ID', --an+'ownerKey',
    owner=tb,
    type='INT'
  };
}
--------------------------------------------------------------------------------
--- creating two Foriegn keyes for each multi-valued attribute
--------------------------------------------------------------------------------
relation mvAttsToFkey {
   an : String;

  checkonly domain OO at : myOO::Attribute {
    name = an
  };

  enforce domain DB tb : myDB::Table {
   fKeys=f1 : myDB::Fkey{fCols=col2:myDB::Column {},ref=ownerTable:myDB::Table{}},
   cols=col2:myDB::Column {name=at.owner.name+'ID'},
   cols=col1:myDB::Column {name=an}
  };

  checkonly domain DB ownerTable : myDB::Table{};
  when {
    mvAttsToCol(at,tb,col1,col2);
  }
  where {
  }
}
--------------------------------------------------------------------
-- creating a table for each multi-valued association
--------------------------------------------------------------------
top relation mvAssociToTable {
  a : String;
  col1:myDB::Column;
  col2:myDB::Column;
  srcTable:myDB::Table;
  trgTable:myDB::Table;

  checkonly domain OO associ : myOO::Association {
    name = a,
    associSchema = cls:myOO::Schema{}
    };
  enforce domain DB tb : myDB::Table {
    name = a,
    schema = dbs:myDB::DBSchema{}
  };
  when {
    (associ.ubound>1 or associ.ubound=-1);
    ClassSchema2DBSchema(cls, dbs);
   ClassToTable(associ.src,srcTable);
   ClassToTable(associ.trg,trgTable);
  }
  where {
   mvAssociToCol(associ,tb,col1,col2);
   mvAssociToFkey(associ,tb,srcTable,trgTable);
  }

}
------------------------------------------------------------
-- creating two col for each multi-valued association.
------------------------------------------------------------
relation mvAssociToCol {
  a : String;

  checkonly domain OO associ : myOO::Association {
    name = a
  };

  enforce domain DB tb : myDB::Table {
  };
  enforce domain DB col1 : myDB::Column {
   name =associ.trg.name+'ID',
   owner=tb,
   type='INT'
  };
  enforce domain DB col2 : myDB::Column {
   name=associ.src.name+'ID',
   owner=tb,
   type='INT'
  };
}
```

```
-------------------------------------------------------------------------
--creating two foreign keys for each multi-valued association
-------------------------------------------------------------------------
relation mvAssociToFkey {
  a : String;

  checkonly domain OO associ : myOO::Association {
    name = a
  };
  enforce domain DB tb : myDB::Table {
   fKeys=f1 : myDB::Fkey{fCols=col1:myDB::Column {},ref=trgTb: myDB::Table {}},
   fKeys=f2 : myDB::Fkey{fCols=col2:myDB::Column {},ref=srcTb: myDB::Table {}}
  };
  checkonly domain DB srcTb : myDB::Table {};
  checkonly domain DB trgTb : myDB::Table {};
  when {
    mvAssociToCol(associ,tb,col1,col2);
  }
}

-----------------------------------------------------
-- creating a column for every Inheritance
-----------------------------------------------------
top relation InherToCol {
  cn : String;
  parentTable:myDB::Table;

  checkonly domain OO cl : myOO::Class {
    name = cn
  };
  enforce domain DB tb : myDB::Table {
   cols=col1:myDB::Column{name = 'parent-'+cl.parent.name+'ID',type='INT'}
  };

  when {
   cl.parent->notEmpty();
   ClassToTable(cl,tb);
   ClassToTable(cl.parent,parentTable);
  }
  where {
   InherToFkey(cl,tb,col1,parentTable);
  }

}
-----------------------------------------------
--creating a FKey for each inheritance
-----------------------------------------------
relation InherToFkey {
  cn : String;

  checkonly domain OO cl : myOO::Class {
    name = cn
  };
  enforce domain DB tb : myDB::Table {
   fKeys=f1 : myDB::Fkey{fCols=col:myDB::Column {},ref=pTable:myDB::Table{}}
  };
  checkonly domain DB col : myDB::Column{};
  checkonly domain DB pTable : myDB::Table {};

}
--------------------------------------------------------------
--crating a column for each single-valued association
--------------------------------------------------------------
top relation svAssociToCol {
  a : String;
  trgTable:myDB::Table;

  checkonly domain OO associ : myOO::Association {
    name = a
  };

  enforce domain DB col1 : myDB::Column {
   name =a+associ.trg.name+'ID',
   owner=srcTable:myDB::Table{}
  };

  when{
    not (associ.ubound>1 or associ.ubound=-1);
    ClassToTable(associ.src,srcTable);
    ClassToTable(associ.trg,trgTable);
  }
  where{
   svAssociToFkey(associ,srcTable,col1,trgTable);
  }
}
--------------------------------------------------------------------
-- crating a foreign key for each single value asociation
--------------------------------------------------------------------
relation svAssociToFkey {
  cn : String;

  checkonly domain OO associ : myOO::Association {
    name = cn
  };
  enforce domain DB tb : myDB::Table {
   fKeys=f1 : myDB::Fkey{fCols=col:myDB::Column {},ref=trgTable:myDB::Table{}}
```

```
      };
      checkonly domain DB col : myDB::Column{};
      checkonly domain DB trgTable : myDB::Table {};

  }

}
```