Decomposition of Relations for Multi-model Consistency Preservation

Master's Thesis of

Aurélien Pepin

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

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Heiko Klare
Second advisor:	Dr.-Ing. Erik Burger
External advisor:	Nils Gesbert (Grenoble INP)

20. May 2019 – 19. November 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 19. November 2019

. . . . . .

(Aurélien Pepin)

Abstract

In the usual software development cycle, the phase of conception involves the creation of models of the software system. These models are developed according to specifications and serve as documentation for software developers. UML diagrams are an example of frequently used models. Model-driven software development is an approach in which models are at the heart of software development. They are an integral part of the software system, each model representing a specific aspect of the system. Models are developed, maintained and tested. Automated or semi-automated model transformations analyze models and generate source code.

When two models represent the same software system, they can share information. In this case, they are said to be *interrelated*. For example, the code and the UML class diagram of an object-oriented software system have class names in common. As a result, modifying a model results in *inconsistencies*. The automated or semi-automated management of inconsistencies is known as *consistency preservation*. Moreover, dependencies between models are called *consistency relations*. These relations are associated with model transformations. When a consistency relation is broken, the related transformation modifies models to repair it.

Multi-model consistency preservation is a recent area of research that focuses on consistency relation networks, i.e. on what happens when several models and consistency relations coexist. Consistency relation networks raise new issues, e.g. when two consistency relations are incompatible. The purpose of this thesis is the design and the implementation of the *decomposition of relations*, an optimization technique for consistency relation networks. The aim of the decomposition procedure is to detect redundant information within consistency relations.

The presented decomposition procedure does not alter consistency specifications. Instead, it returns a decomposition of the specification, i.e. a set of independent and non-redundant consistency relations. This decomposition facilitates the detection of possible contradictory consistency relations. As a result, the decomposition procedure helps the developer to find incompatibilities in consistency specifications.

We provide an evaluation of our approach by assessing the functional correctness of the procedure and the applicability of its prototype. The evaluation shows that the use of the procedure in a consistency preservation process is beneficial.

Keywords: software engineering, model-driven software development, model transformations, decomposition of relations, multi-model consistency.

Zusammenfassung

Im üblichen Softwareentwicklungsprozess beinhaltet die Design-Phase die Erstellung von Modellen des Softwaresystems. Diese Modelle werden nach bestimmten Anforderungen entwickelt und dienen als Dokumentation für Softwareentwickler. UML-Diagramme sind ein Beispiel für häufig verwendete Modelle. Modellgetriebene Softwareentwicklung ist ein Ansatz, bei dem Modelle im Mittelpunkt der Softwareentwicklung stehen. Sie werden ein wichtiger Bestandteil des Softwaresystems, dadurch dass jedes Modell einen bestimmten Aspekt des Systems darstellt. Modelle werden entwickelt, gewartet und getestet. Automatisierte oder teilautomatisierte Modelltransformationen analysieren Modelle und erzeugen Quellcode.

Wenn zwei Modelle das gleiche Softwaresystem abbilden, können sie Informationen miteinander gemein haben. In diesem Fall werden sie als *zusammenhängend* bezeichnet. Beispielsweise haben der Code und das UML-Klassendiagramm eines objektorientierten Softwaresystems gemeinsame Klassennamen. Infolgedessen führt die Änderung eines Modells zu *Inkonsistenzen.* Die automatisierte oder teilautomatisierte Auflösung von Inkonsistenzen wird als *Konsistenzhaltung* bezeichnet. Darüber hinaus werden Abhängigkeiten zwischen Modellen als *Konsistenzrelationen* bezeichnet. Diese Relationen sind mit Modelltransformationen assoziiert. Falls eine Konsistenzrelation verletzt wird, modifiziert die assoziierte Transformation Modelle, um die Konsistenzrelation zu reparieren.

Multi-Modell-Konsistenzhaltung ist ein entstehendes Forschungsgebiet, das sich auf Netzwerke von Konsistenzrelationen bezieht, also darauf, was passiert, wenn mehrere Modelle und Konsistenzrelationen gleichzeitig funktionieren. Netzwerke von Konsistenzrelationen werfen neue Problem auf, zum Beispiel wenn zwei Konsistenzrelationen nicht kompatibel sind. Der Zweck dieser Masterarbeit ist das Design und die Implementierung der *Dekomposition von Relationen*, einer Optimierungstechnik für Netzwerke von Konsistenzrelationen. Ziel des Dekompositionsverfahrens ist die automatische Erkennung redundanter Informationen innerhalb von Konsistenzrelationen.

Das vorgestellte Dekompositionsverfahren verändert Konsistenzspezifikationen nicht. Stattdessen gibt es eine Dekomposition der Spezifikation aus. Eine Dekomposition ist eine Menge von unabhängigen und nicht redundanten Konsistenzrelationen. Diese Dekomposition erleichtert die Erkennung möglicher widersprüchlicher Konsistenzrelationen. Dadurch hilft das Dekompositionsverfahren dem Entwickler, Inkompatibilitäten in Konsistenzspezifikationen zu finden.

Der Ansatz dieser Arbeit wird durch die funktionale Korrektheit des Verfahrens und die Anwendbarkeit des Prototyps evaluiert. Die Evaluation zeigt, dass die Verwendung des Verfahrens in einem Konsistenzhaltungsprozess von Vorteil ist.

Stichwörter: Softwaretechnik, modellgetriebene Softwareentwicklung, Modelltransformationen, Dekomposition von Relationen, Multi-Modell-Konsistenzhaltung.

Résumé

Dans le cycle de développement habituel d'un logiciel, la phase de conception comprend la création de modèles du système logiciel. Ces modèles sont mis au point selon un cahier des charges et servent de documentation pour le développement du logiciel. Les diagrammes UML sont un exemple de modèles fréquemment utilisés. L'ingénierie dirigée par les modèles est une approche dans laquelle les modèles sont au cœur du développement. Ils font partie intégrante du système logiciel : chaque modèle est une représentation d'un aspect du système logiciel. Les modèles sont développés, fréquemment mis à jour et testés. Des transformations de modèles (automatiques ou semi-automatiques) analysent les modèles et génèrent du code, le code devenant alors aussi un modèle du logiciel.

Lorsque deux modèles représentent un même système logiciel, ils contiennent de l'information en commun. Ils sont dits *interdépendants*. Par exemple, le code et le diagramme UML de conception d'un système orienté objet ont entre autres en commun les noms des classes qu'ils contiennent. Dans ce cas, modifier un modèle génère des incohérences. La résolution automatique ou semi-automatique de ces incohérences est connue sous le nom de *préservation de la cohérence*. De même, les dépendances entre les modèles sont appelées *relations de cohérence*. Les relations de cohérence sont associées à des transformations de modèles. Lorsqu'une relation de cohérence est rompue, la transformation liée modifie les modèles afin de la restaurer.

La *préservation de la cohérence multi-modèles* est un domaine de recherche récent dans lequel on s'intéresse aux réseaux de relations de cohérence, c'est-à-dire à ce qu'il se passe lorsque plusieurs modèles et relations de cohérence coexistent. Les réseaux de relations de cohérence soulèvent de nouveaux problèmes, par exemple lorsque deux relations de cohérence sont incompatibles. L'objectif de ce mémoire est la conception et l'implémentation de la *décomposition de relations*, une technique d'optimisation pour les réseaux de relations de cohérence. Le but de la décomposition de relations est d'identifier automatiquement les informations redondantes au sein des relations de cohérence afin d'en accroître la compatibilité.

La procédure de décomposition présentée n'altère pas les spécifications de cohérence. Elle renvoie plutôt une décomposition de la spécification, c'est-à-dire un ensemble de relations de cohérence indépendentes et non redondantes. Cette décomposition facilite la détection de relations de cohérence possiblement contradictoires. Par conséquent, la procédure aide le développeur à trouver des incompatibilités dans les spécifications de cohérence. L'évaluation de cette approche est fondée sur la correction de la procédure et sur l'applicabilité du prototype qui en résulte. Cette évaluation montre que l'intégration de la procédure dans un processus de préservation de la cohérence est bénéfique.

Mots-clés : génie logiciel, ingénierie dirigiée par les modèles, transformation de modèles, décomposition de relations, cohérence multi-modèles.

Contents

Abstract i

Zusammenfassung iii

Résumé v

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Goal of the thesis . . . 2
- 1.3. Structure of the thesis . . . 3

2. Foundations 5
- 2.1. Models, Metamodels, Model Transformations . . . 5
  - 2.1.1. Models . . . 5
  - 2.1.2. Metamodels . . . 6
  - 2.1.3. Model Transformations . . . 6
- 2.2. Model-Driven Software Development . . . 8
  - 2.2.1. The Ecore Meta-metamodel . . . 8
  - 2.2.2. The Object Constraint Language . . . 9
  - 2.2.3. Transformation Languages . . . 10
- 2.3. Formal Foundations of Models . . . 11
  - 2.3.1. Formal Metamodels . . . 11
  - 2.3.2. Formal Models and Instances . . . 12
- 2.4. Model Consistency Preservation . . . 13
  - 2.4.1. Consistency Relations . . . 13
  - 2.4.2. Model Transformations for Consistency Relations . . . 13
  - 2.4.3. Multi-Model Consistency Preservation . . . 15
- 2.5. Constraint Satisfaction . . . 16
  - 2.5.1. Constraint Networks . . . 17
  - 2.5.2. Constraint Graphs and Hypergraphs . . . 17
- 2.6. Automated Deduction . . . 18
  - 2.6.1. First-Order Logic . . . 18
  - 2.6.2. Satisfiability Modulo Theories . . . 20

3. Consistency Preservation 21
- 3.1. Description of Consistency Relations . . . 21
  - 3.1.1. Consistency Relation Graph . . . 21
  - 3.1.2. Consistency Rule . . . 22

3.1.3. Consistency Specification . . . 25
3.2. Consistency with QVT-R . . . 25
3.3. Structure of a QVT-R Specification . . . 26
3.3.1. Imports . . . 27
3.3.2. Relational Transformations . . . 27
3.3.3. Relations . . . 27
3.3.4. Relation Domains . . . 28
3.3.5. Expressions and Conditions . . . 29
3.4. From QVT-R to Consistency Rules . . . 30
3.4.1. From Domain Pattern to Condition on a Metaclass . . . 31
3.4.2. From Domain to Condition on a Metaclass Tuple . . . 32
3.4.3. From Transformation to Consistency Rule . . . 33

4. Principles of Decomposition **35**
4.1. Introduction to the Decomposition Procedure . . . 35
4.1.1. Equivalent Consistency Specifications . . . 36
4.1.2. Complexity of Consistency Specifications . . . 36
4.2. Means of Decomposition of Specifications . . . 38
4.2.1. Independent Consistency Subgraphs . . . 39
4.2.2. Totally Redundant Consistency Relations . . . 39
4.2.3. Partially Redundant Consistency Relations . . . 41
4.2.4. Towards a Decomposition Procedure . . . 43
4.3. Formal Properties . . . 44
4.3.1. Conservativeness . . . 44
4.3.2. Usefulness . . . 45

5. Decomposition Procedure **49**
5.1. Tractable Consistency Relations . . . 49
5.1.1. Two Aspects of Consistency Specifications . . . 49
5.1.2. Metagraph . . . 51
5.1.3. Metagraphs and Constraint Networks . . . 52
5.2. Outline of the Decomposition Procedure . . . 53
5.3. From Consistency Specification to Metagraph . . . 53
5.3.1. Inputs of the Procedure . . . 56
5.3.2. Recursive Construction of QVT-R Concepts . . . 58
5.3.3. Translation of Global Aspects of Specifications . . . 60
5.3.4. Translation of Local Aspects of Specifications . . . 64
5.4. From Metagraph To Decomposition . . . 69
5.4.1. Metagraph Dual . . . 69
5.4.2. Independent Subsets of Meta-Edges . . . 71
5.4.3. Generation of Combinations of Meta-Edges . . . 74
5.4.4. Detection of Redundant Rules . . . 77

6. Constraint Translation **83**
6.1. Symbolic Computation for OCL and QVT-R . . . 83
6.1.1. Automation of the Decomposition Procedure . . . 83
6.1.2. Choosing an Approach for Constraint Translation . . . 84
6.1.3. Theorem Proving for Decomposition . . . 86
6.2. Primitive Datatypes . . . 86
6.3. Data Structures . . . 87
6.3.1. Collection Literals . . . 87
6.3.2. Collections from Role Names . . . 88
6.4. Operations . . . 89
6.4.1. Arithmetic Operations . . . 89
6.4.2. Boolean Operations . . . 90
6.4.3. Conversion Operations . . . 90
6.4.4. Equality Operators . . . 90
6.4.5. Order Relations and Extrema . . . 90
6.4.6. Collections Operations . . . 90
6.4.7. String Operations . . . 91
6.4.8. Untranslatable Operations . . . 91

7. Evaluation **93**
7.1. Methodology . . . 93
7.1.1. Addressing Research Questions . . . 93
7.1.2. Evaluation Material . . . 94
7.2. Functional Correctness . . . 94
7.2.1. Finding Existing Tree-Like Specifications . . . 95
7.2.2. Unaltered Consistency Specifications . . . 97
7.3. Applicability . . . 99
7.3.1. Example Scenarios . . . 100
7.3.2. Execution Results . . . 100
7.3.3. Threats to Validity . . . 103
7.4. Discussion and Further Evaluation . . . 103
7.4.1. Benefits . . . 103
7.4.2. Limitations . . . 105
7.4.3. Further Evaluation . . . 106

8. Related Work **107**
8.1. Model Consistency Preservation . . . 107
8.1.1. Approaches for Consistency . . . 107
8.1.2. Multi-Model Consistency Preservation . . . 109
8.1.3. Model Transformation Decomposition and Composition . . . 110
8.2. Formalization of QVT-R . . . 110
8.3. Formal Techniques for Transformation Languages . . . 111
8.3.1. Automated Techniques . . . 111
8.3.2. Interactive Techniques . . . 112
8.3.3. Model Finding . . . 112

9. Conclusion and Future Work **113**
9.1. Conclusion . . . 113
9.2. Future Work . . . 114
9.2.1. Extension to Other Constructs . . . 114
9.2.2. Extension to Other Symbolic Computation Tools . . . 114
9.2.3. Extension to Other Contexts . . . 114

Bibliography **117**

A. Appendix: Translation of OCL Operations **131**
A.1. Arithmetic Operations . . . 131
A.2. Boolean Operations . . . 132
A.3. Conversion Operations . . . 132
A.4. Equality Operators . . . 132
A.5. Order Relations and Extrema . . . 133
A.6. Collection Operations . . . 133
A.6.1. Operations For Collections . . . 133
A.6.2. Operations For Sequences . . . 134
A.6.3. Operations For Sets . . . 134
A.7. String Operations . . . 135

1. Introduction

1.1. Motivation

In engineering, *modeling* is a common approach to deal with complex systems. Modeling is defined as the efficient use of models, i.e. simplified representations of an aspect of a system, for a given objective [Sel03]. For example, meteorologists design mathematical models of the weather for prediction purposes.

In software engineering, modeling focuses on software systems. It aims to represent different properties of the system using models of architecture, reliability, performance, security, etc. *Model-driven software development* suggests to use these models as parts of the software to increase both speed and quality of software development [VS06].

As models of a system describe aspects of the same system, they contain overlapping and dependent information. They are interrelated. For instance, the code and the design model (e.g., a UML class diagram) of an object-oriented software system will probably have class names in common. Because they are fundamental and primary artifacts in model-driven software development, models are frequently modified. Modifying a model leads to *inconsistencies*. Solving these inconsistencies through automated mechanisms is known as *consistency preservation*. We refer to dependencies between models as *consistency relations*. The most common way to restore consistency relations is to use *bidirectional transformations* (bx) [Ste08], a special case of model transformations. Model transformations correspond to automatic generations of target models from source models. When consistency relations hold for pairs of models, such transformations are called binary transformations. Moreover, transformations for consistency preservation are usually *incremental*, so that specific parts of models are updated rather than recreating models each time.

As stated in [Kla18], it is also possible to define *multi-model consistency relations*, i.e. consistency relations between more than two models. Related transformations are known as *multiary bx*. Stevens proved in [Ste17a] that under some reasonable conditions, multiary bx can be defined in terms of binary bx. That is, multi-model consistency preservation could be achieved by combining transformations for binary consistency relations. A specification for multi-model consistency can be regarded as a network of metamodels linked by consistency relations.

This approach has several advantages. One of these advantages is that it allows to develop independent binary transformations rather than complex n-ary transformations. In practice, this corresponds to the common case in which a developer does not master all possible modeling languages. A consistency specification developed according to this approach is easier to maintain as transformations do not need to know each other to operate

together. However, this approach introduces other problems such as interoperability issues between transformations. Problems and proposed solutions are discussed in detail in [Kla18]. One of these solutions is the *decomposition of consistency relations*.

1.2. Goal of the thesis

The goal of this thesis is to investigate the decomposition of consistency relations, an approach introduced in [Kla18] for optimizing the applicability of consistency specifications. In this thesis, we aim to provide an implementation of this optimization technique called the *decomposition procedure*.

The main motivation behind decomposition is that in a network of consistency relations, combinations of various consistency relations may ultimately play the same role in consistency preservation. In other words, what a consistency relation preserves between two metamodels may be (at least partly) preserved by an alternative combination of relations between the same metamodels. In such a case, the relation can be decomposed.

For example, let a, b and c be three attributes in three different metamodels. Suppose that consistency is preserved if all three attributes have the same value. This requirement can be represented by three consistency relations: $a = b$, $b = c$ and $a = c$. If the value of an attribute is updated, the value of the other two must be updated accordingly. To this purpose, two consistency relations are already sufficient. For example, the combination of $a = b$ and $b = c$ ensures that $a = c$. Thus, one of the three relations can be replaced by the combination of the other two. This is an example of decomposition.

An important requirement for consistency specifications to be applicable is that they contain no contradictory consistency relations. Two relations are contradictory if no model can fulfill both relations at the same time. A benefit of the decomposition procedure is that it can be used to find possibly contradictory relations. The decomposition procedure removes and separates as many relations as possible. If there are two distinct combinations of relations between two metamodels after the decomposition, this means that no combination can replace the other. In other words, there may be contradictory relations in these combinations. The decomposition procedure makes it easier to detect possibly contradictory relations, thus indirectly improving the applicability of specifications.

In this thesis, we first give a formal meaning to the concept of consistency specification and we provide a concrete way to write specifications. This is achieved by using QVT-R, a language to write model transformations, for which we show that it can be used to write a consistency specification. We then describe the principles of decomposition of consistency relations and we also discuss the benefits of decomposition.

With consistency specifications written in QVT-R, we present an implementation of the decomposition procedure, i.e. a tool that takes a consistency specification as an input and returns another simplified specification that preserves consistency in the same way. In summary, this thesis has the following research goal:

> Given a consistency specification, identify decomposable relations, i.e. relations that can be replaced by an alternative combination of consistency relations without altering the specification. Provide a way to do it systematically by analyzing QVT-R transformations that represent consistency specifications.

To meet this goal, this thesis answers the following research questions:

Q1. What does it mean for a consistency relation to be decomposable?

Q2. How can QVT-R be used to specify consistency in a set of metamodels?

Q3. How to design a decomposition procedure that:

- decomposes consistency relations in an optimal way,
- and ensures that the consistency specification is not altered?

1.3. Structure of the thesis

First, Chapter 2 presents some general concepts that lay the foundation of this thesis. In particular, important concepts of model-driven software development and symbolic computation are introduced.

Chapter 3 gives a formal meaning to the notion of consistency. It introduces QVT-R, a transformation language, and shows how the main concepts of this language can be compared with the theoretical framework for consistency.

Chapter 4 presents the characteristics of the decomposition procedure. This involves a description of the input and the output of the algorithm, as well as the purpose of decomposition. Some methods to achieve a decomposition of consistency relations are also introduced. Finally, important properties that the decomposition procedure must meet are discussed and formalized.

Chapter 5 is a detailed explanation of the implementation of the decomposition procedure. It provides an algorithmic way to implement decomposition methods mentioned in the previous chapter. The chapter is based on the structure of metagraph, a comprehensive and usable representation of consistency specifications.

Chapter 6 describes the use of an automated theorem prover to achieve the decomposition of relations. In particular, it shows how parts of the QVT-R transformation language can be translated into first-order formulae.

Chapter 7 focuses on the evaluation of the proposed decomposition procedure. We mainly evaluate the functional correctness of the procedure and its applicability in the context of multi-model consistency preservation. Then, we discuss the benefits and the limitations of such an approach.

Finally, Chapter 8 gives an overview of the existing work in consistency preservation. It also surveys the use of formal methods in the field of model transformations.

2. Foundations

This thesis is based on models and relations between them. Models are fundamental in science in general. The design of models, namely *scientific modeling*, is a useful tool to deal with complexity of systems. In general terms, models are simplified representations of a system or a process used to describe and predict phenomena from real life. We focus on software modeling, a case of modeling where the system is a software. Moreover, this thesis uses constraint satisfaction and automated deduction to reason about models.

This chapter presents all important concepts in software modeling and automated deduction that serve as a basis for this thesis. In particular, the first section introduces important notions of model theory: models, metamodels and model transformations. The second section is an overview of technologies of model-driven software development, a way to use software modeling results as parts of a software. The third section sets out the foundations of consistency preservation, a challenge that arises when working with multiple models. The fourth section is an introduction to important techniques in the field of constraint satisfaction. Finally, the fifth part presents notions of automated deduction.

2.1. Models, Metamodels, Model Transformations

2.1.1. Models

2.1.1.1. Features of Models

Software modeling is usually achieved using modeling languages, i.e. languages to design models. The *Unified Modeling Language* (UML) is a well-known example of general-purpose modeling language [RJB04].

An appropriate definition of models in software engineering was proposed by Stachowiak in [Sta73]. Models can be identified by their features. Stachowiak found three of them that every model has.

- *Representation feature.* Each model is a model of something, i.e. a model is always associated to an *original*.
- *Reduction feature.* A model never captures all aspects of an original, but only those that are relevant in the context of the modeling.
- *Pragmatic feature.* A model is only valid in a given context, especially for particular subjects and intervals of time.

For example, a UML class diagram is a graphical model of a software system, which represents the object-oriented structure of the system. Among other uses, it is provided to developers during the implementation phase of the software.

2.1.1.2. Originals

The original is what the model represents. Generally, an original can be represented by multiple models describing different aspects of it. For example, a software system can have architectural models, performance models, security models, etc. Conversely, a model can also be used for several originals. The relation from an original to a model is called *abstraction*, whereas the relation from a model to an original is called *instanciation*.

2.1.2. Metamodels

A metamodel is a model of a model, i.e. a model that describes the structure of models [VS06]. Following the definitions of originals, a metamodel represents a model whereas a model instantiates a metamodel. In theory, it is possible to define an infinite sequence of abstractions since every model can be itself modeled. In practice, the most abstract metamodel is *self-descriptive*, i.e. it is able to describe itself.

Metamodels are fundamental in modeling languages. For example, valid models according to the specification of UML can be seen as valid *instances* of the UML metamodel. The standard to which the metamodel that describes the UML language conforms is called the *Meta-Object Facility* (MOF) [Obj16a]. The MOF standard can itself be regarded as a model for metamodels, i.e. a meta-metamodel.

The Meta-Object Facility is designed as an architecture of four *modeling layers*. These layers range from `M0` to `M3`. More precisely, layers are defined as follows:

- `M0` is the original, i.e. an object in the reality;
- `M1` is the (user) model that represents `M0`;
- `M2` is the metamodel, e.g. UML, whose instances are `M1` models;
- `M3` is the self-descriptive MOF meta-metamodel to which `M2` conforms.

In practice, this number of layers is sufficient for a large number of applications. Most of modeling languages are defined at the `M2` layer, i.e. as instances of the meta-metamodel provided by the MOF standard.

This is the case with the metamodel defined by the UML standard [Obj16d], but also with specific metamodels such as the *Palladio Component Model* (PCM, for performance prediction) [BKR09] or the *AUTomotive Open System ARchitecture* (AUTOSAR, for automotive electronics) [Für+09].

2.1.3. Model Transformations

2.1.3.1. Principles of Transformations

The systematic and sometimes automated use of models in software engineering involves the creation of tools to manipulate them and to make them work together. This is the purpose of *model transformations*. As stated in [Kle+03], a model transformation is an automatic generation of a target model from a source model, according to a set of transformation

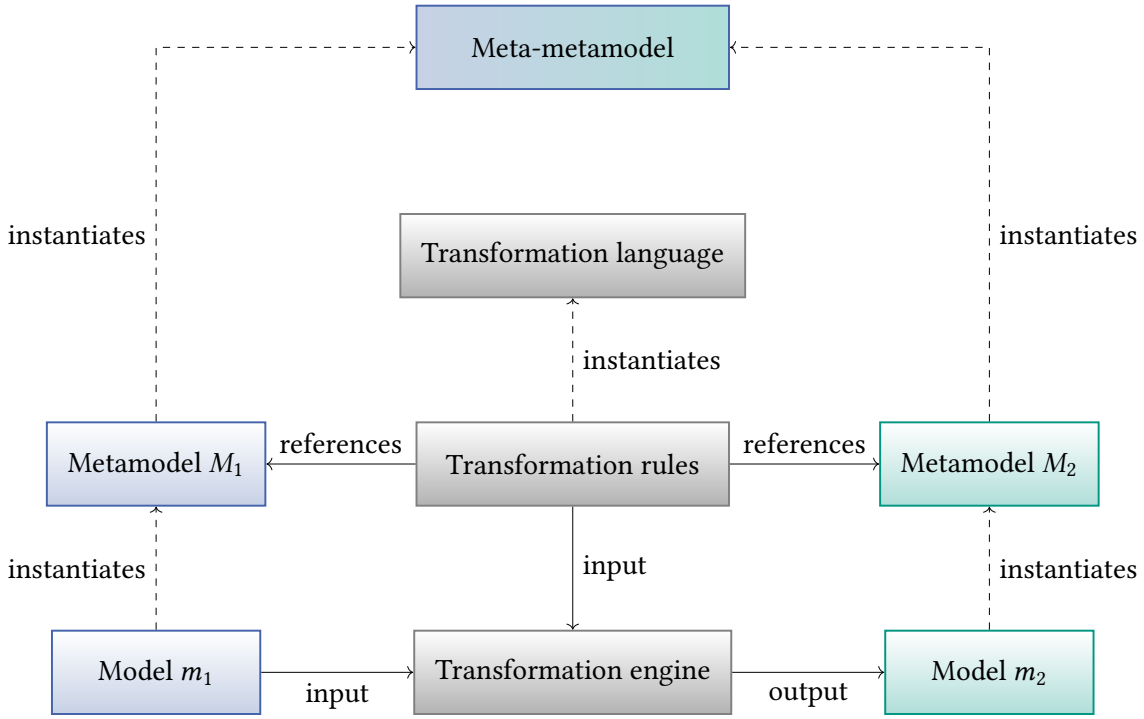


Figure 2.1.: Principles of model transformations

rules. In particular, one purpose of model transformations is to generate platform-specific models and code from high-level models [SK03].

Figure 2.1 depicts the use of model transformations. First, *transformations rules* are specified between two metamodels M_1 and M_2. To be processed automatically, these rules obey to a certain syntax defined by a *transformation language*. Transformation languages are programming languages in which it is easy to refer to metamodel elements. Second, transformations are applied on models through a *transformation engine* that interprets the transformation rules. Suppose that m_1 (resp. m_2) is an instance of M_1 (resp. M_2) and that there exist transformations rules referencing M_1 and M_2. The transformation engine takes m_1 and the set of rules as inputs in order to create (or update) the model m_2.

2.1.3.2. Classification of Model Transformations

Classification of model transformations encompasses many criteria [Ste08]; [MV06]. For example, model transformations may be either automated or interactive. If transformations relate models of the same modeling layer, they are said to be horizontal; vertical otherwise. If transmations relate instances of the same metamodel, they are called endogenous; exogenous otherwise. For example, code refactoring is achieved through horizontal and endogen model transformations.

Another criterion is the direction of transformations. If models related by a transformation are either sources or targets (but not both), the transformation is called unidirectional. Otherwise, if models can be both sources and targets, the transformation is called bidi-

rectional. *Bidirectional model transformations* form an important and well-researched category of model transformations [Ste17a].

2.2. Model-Driven Software Development

The approach in which models and model transformations are at the heart of the construction of a software system is called *model-driven software development* (MDSD). Models are said to be *primary artefacts* as they can't be put aside during the development, unlike the usual development approaches in which they are mainly used for documentation.

The promise of model-driven software development is to focus on domain-specific challenges using *domain-specific languages* (DSL). Platform-specific challenges (including code generation) could then be performed by model transformations. The objective of this approach includes better interoperability, platform independence and better maintainability (through reduced redondancy) [VS06].

2.2.1. The Ecore Meta-metamodel

Tools for model-driven software development are not fully mature yet [KMT12]. However, there already exist multiple initiatives to foster the use of models and metamodels in software engineering. One of the most advanced ecosystem to this end is Eclipse[1]. Modeling tools provided by Eclipse are part of the *Eclipse Modeling Framework*[2] (EMF). This framework provides an implementation of model-driven engineering standards defined by the *Object Management Group*[3] (OMG).

The OMG is a consortium whose purpose is to promote object modeling. The model-driven engineering standards it defines are integrated into an approach called *Model-Driven Architecture* (MDA) [Po01]. The MDA approach is entirely based on the Meta-Object Facility. As a result, the Eclipse Modeling Framework provides a support for the MOF framework and other technologies that refer to it.

The most important part of the MOF framework is its meta-metamodel. This is the starting point for modeling metamodels such as that of UML and other modeling or transformation languages. The MOF meta-metamodel comes with two *compliance points* that describe depending on the level of detail required to use the meta-metamodel: EMOF (*Essential MOF*, a subset of MOF for facilities of object-oriented programming and XML) and CMOF (*Complete MOF*, built from EMOF and UML constructs).

Ecore is a meta-metamodel defined in EMF that can be regarded as an implementation of EMOF. As a consequence, Ecore is compatible with standard derived from the MOF meta-metamodel. Like the MOF meta-metamodel, Ecore is self-descriptive. The kernel of Ecore is represented in Figure 2.2 [Ste+08]. It is actually close to the formalism of UML: the most fundamental element is the class (`EClass`). Each class can have attributes (`EAttribute`) and references (`EReference`). A hierarchy of classes, similar to generalization in UML, is available through `eSuperTypes` references.

[1] `https://www.eclipse.org`
[2] `https://wiki.eclipse.org/Eclipse_Modeling_Framework`
[3] `https://www.omg.org`

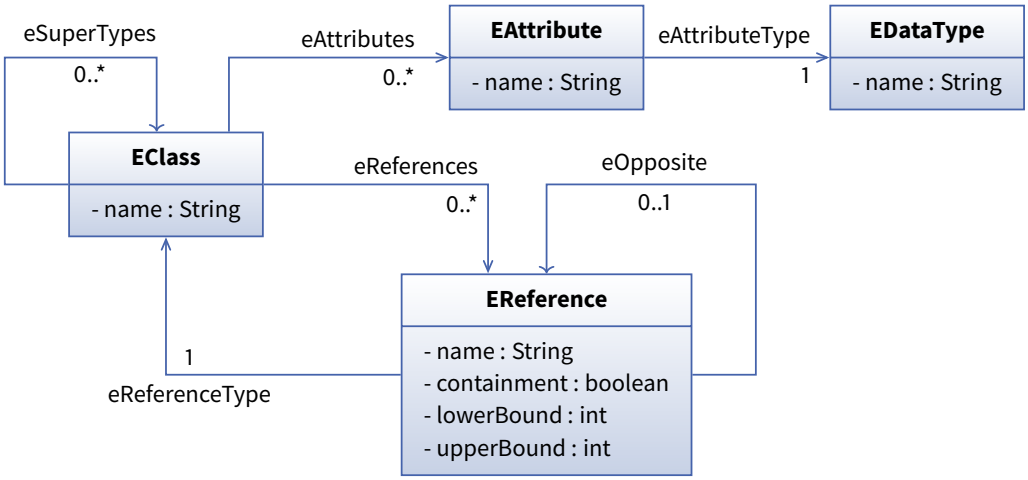


Figure 2.2.: Kernel of the Ecore meta-metamodel

The metamodel is self-descriptive in that it is itself made up of classes with attributes and references. Given that the metamodel of the UML standard can be regarded as an instance of the Ecore meta-metamodel, a UML class with an attribute can be represented as an instance of `EClass` that references an instance of `EAttribute`.

Ecore also defines primitive data types that corresponds to usual datatypes in programming languages, e.g. `EInt`, `EBool`, `EString`, etc. These data types can easily be mapped to primitive data types of programming languages such as Java. Altogether, metamodels of any nature can be represented as instances of the Ecore meta-metamodel. They are usually serialized in `*.ecore` files using an extension of XML called *XML Metadata Interchange* (XMI).

2.2.2. The Object Constraint Language

The representation of metamodels as instances of the Ecore meta-metamodel is purely syntactic, in the sense that it is only a set of elements and relationships between them. However, this representation is incomplete. For example, there is no way to indicate syntactically that an instance of an *EAttribute* takes a restricted set of values.

To overcome this limitation, the MDA approach provides a standard to specify the semantics of metamodels called the *Object Constraint Language* (OCL) [WK03]. It is a declarative and strongly typed language that provides constraints and invariants for MOF models (including UML models). This is a convenient way to express additional rules on models such as semantic constraints.

Constraints in OCL usually consists of two parts. First, a *context* representing a class. It indicates on which class the constraint should apply. Second, the content of the constraint expressed with a boolean OCL expression. The expression must evaluate to TRUE for an object (i.e. a class instance) to fulfill the constraint. OCL also includes primitive data types that can be mapped to those of Ecore.

Due to its ability to relate metamodel elements and its high expressiveness, OCL is embedded in many transformation languages. In particular, other languages often embed

Essential OCL, the minimal subset of OCL required to work with EMOF, or *Imperative OCL*, a Turing-complete extension of OCL with side effects [Obj16c].

2.2.3. Transformation Languages

As stated in Section 2.1.3.2, there are many criteria classify model transformations. Consequently, there exist many transformation languages with different features. The MDA approach provides its own set of transformation languages called *Query/View/Transformation* (QVT) [Obj16b].

As with usual programming languages, transformation languages can be divided into two main paradigmes: *imperative* languages and *declarative* languages. Imperative transformation languages describe *how* the transformation should be performed whereas declarative languages focus on *what* transformations should perform. Consequently, QVT defines two main languages:

- QVT Relations (QVT-R) is a declarative language. It allows the specification of unidirectional and bidirectional transformations. QVT-R transformations can be regarded as sets of conditions on models. The execution of transformations can impose the fulfilement of these conditions by updating models.
- QVT Operational (QVT-O) is an imperative language. It is useful when a specification of steps is required to derive the target model from the source model. It only allows the specification of unidirectional transformations.

In addition to these languages, QVT Core (QVT-C) is another minimal, declarative language. However, it is rarely used by end-users because QVT-R is declarative too and more expressive. The QVT standard defines a way to transform QVT-R transformations into QVT-C transformations. Figure 2.3 depicts the dependencies resulting from the organization of QVT languages into packages. First, the three languages depend (directly or not) on the EMOF package as they can be expressed as instances of the EMOF meta-metamodel. Second, common concepts between languages have been into several intermediate packages. The first one, QVT Base, consists of constructs that appear both in QVT-C and QVT-R. The second one, QVT Template, is only used by QVT-R. It is made up of all constructs that allow QVT-R to express conditions on models as OCL expressions. Finally, it should be noted that QVT-O uses its own representation of OCL since it is an imperative language that needs side effects.

Semantics of languages of the QVT standard are hard to understand and incomplete [Ste10]. For this reason, their implementation is still under development. While there exists a stable engine for QVT-O, the support of QVT-R is partial at the time of writing of this thesis. Other languages, partly based on QVT, provide a better support.

One of these languages is the *ATLAS Transformation Language* (ATL) [Jou+06]. ATL is an hybrid transformation language: it contains both declarative and imperative features. It has its own meta-metamodel called *Kernel Meta-metamodel* (KM3) but can be used within EMF. Like QVT-R and QVT-O, it uses OCL to manipulate metamodel elements.

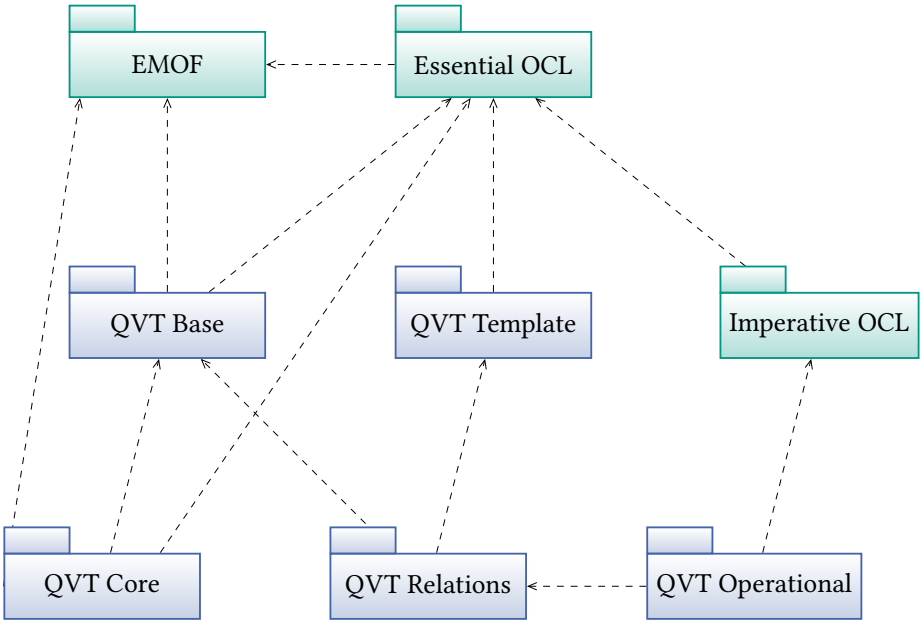


Figure 2.3.: Dependencies between QVT packages

2.3. Formal Foundations of Models

Although models and metamodels are intuitive and proven concepts in software engineering, we now aim to give a formal meaning to them. The point of formalizing concepts of model-driven software development is to use them as objects that can be built, compared and embedded into a theoretical framework to define consistency.

Making model-driven engineering concepts and technologies formal is an efficient way to separate tools from their implementations and to point out more limitations and semantics issues in standards. In particular, most of technologies presented in this thesis are MOF-based, i.e. they have been formalized as instances of the MOF meta-metamodel. See for example the standardisation of QVT [Obj16b], OCL [Obj16c] and UML [Obj16d].

Definitions presented in this section are based on set theory and adapted from the theoretical framework of Kramer [Kra17]. This framework is itself based on the MOF representation of metamodels and models. As a result, the correspondence between these definitions and the representation of models in the MOF standard is straightforward.

2.3.1. Formal Metamodels

We first give a formal meaning to the concept of metamodel.

Definition 2.3.1 *A* **metamodel** M *is a tuple* $(\mathbb{C}, \prec, \mathbb{R}, \mathbb{A}, \mathbb{V})$ *such that* $\mathbb{C}$ *is a set of metaclasses,* $\prec$ *is a partial order on* $\mathbb{C}$ *(i.e.* $\prec \subset \mathbb{C} \times \mathbb{C}$*),* $\mathbb{R}$ *is a set of references of metaclasses,* $\mathbb{A}$ *is a set of attributes of metaclasses and* $\mathbb{V}$ *is a (possibly infinite) set of attribute values.*

Basic components of a metamodel are metaclasses, i.e. classes at the metamodel layer, attributes and references. This perspective can be compared to the Ecore meta-metamodel

as it uses the same concepts. Note that the set of admissible values for attributes is fixed in the metamodel. Then, attributes in models that instantiate metamodels take values in this set of admissible values. Moreover, the $\prec$ order represents the generalization of metaclasses, i.e. `eSuperTypes` in Ecore. For two metaclasses $\mathbb{c}_1, \mathbb{c}_2 \in \mathbb{C}$, $\mathbb{c}_1 \prec \mathbb{c}_2$ means that $\mathbb{c}_2$ is a superclass of $\mathbb{c}_1$. Also, attributes and references can directly be accessed from metamodels rather than being confined to metaclasses in the formalism of Kramer. The reason for this is that it is then easier to access attributes or references of superclasses of a given class. We also need to define what a metaclass tuple is.

Definition 2.3.2 *Let $M = (\mathbb{C}, \prec, \mathbb{R}, \mathbb{A}, \mathbb{V})$ be a metamodel. A* **metaclass tuple** *of M is a tuple $(\mathbb{c}_{i_1}, \ldots, \mathbb{c}_{i_n})$ such that $\mathbb{c}_{i_1}, \ldots, \mathbb{c}_{i_n} \in \mathbb{C}$.*

In other words, a metaclass tuple is a sequence of metaclasses that all belong to the same metamodel. A metaclass can appear multiple times in the tuple and the order of the metaclass tuple is arbitrary.

2.3.2. Formal Models and Instances

We then formalize what happens when a metamodel is instantiated. First, we give a definition of objects, i.e. elements of models and instances of metaclasses.

Definition 2.3.3 *Let $\mathbb{c}$ be a metaclass of a metamodel $M = (\mathbb{C}, \prec, \mathbb{R}, \mathbb{A}, \mathbb{V})$. We say that an* **object** *o instantiates $\mathbb{c}$ if it has links to other objects for references of $\mathbb{c}$ and for references of direct or indirect superclasses of $\mathbb{c}$ and if it has label values for attributes of $\mathbb{c}$ and for attributes of direct or indirect superclasses of $\mathbb{c}$.*

In other words, an object is defined as an abstract element that meets the constraints (attributes, references, superclasses) of the metaclass it instantiates. The set of objects that instantiate a metaclass $\mathbb{c}$ is denoted $\mathcal{I}(\mathbb{c})$.

Definition 2.3.4 *Let $M = (\mathbb{C}, \prec, \mathbb{R}, \mathbb{A}, \mathbb{V})$ be a metamodel where $\mathbb{C} = \{\mathbb{c}_1, \ldots, \mathbb{c}_n\}$. A* **model** *$m$ that instantiates the metamodel M is a tuple $m = (O_{\mathbb{c}_1}, \ldots, O_{\mathbb{c}_n}, \textsc{link}, \textsc{label})$ such that $O_{\mathbb{c}_i}$ is a set of objects that instantiate the metaclass $\mathbb{c}_i$, $\textsc{link} : O \times \mathbb{R} \to \mathcal{P}(O)$ is a function that links objects through references and $\textsc{label} : O \times \mathbb{A} \to \mathcal{P}(\mathbb{V})$ is a function that links an attribute of an object to a value.*

A model is defined as a container for objects with a description of references between these objects (LINK) and a description of values of attributes (LABEL). The model is made up of sets of objects because a metaclass can be instantiated multiple times in a single model, i.e. $O_{\mathbb{c}} \subseteq \mathcal{I}(\mathbb{c})$. Objects by themselves are only of interest when they are combined with other objects and defined in a model. For this reason, links and labels are defined as a property of models, not as a property of objects.

Note that a model m of a metamodel M does not imply that m conforms to M. The reason for this is that LINK and LABEL are not restrictive enough. They can relate arbitrary

objects, references and attributes. For example, LABEL can relate an attribute with an object that does not contain it. Kramer defines the *conformance* of a model by applying restrictions on LINK and LABEL [Kra17, p. 43]. From now on, we only consider models of a metamodel that conform to it. These conforming models are simply referred to as models. The set of models of a metamodel M (also called the universe of M) is denoted $\mathcal{I}(M)$.

2.4. Model Consistency Preservation

Keeping all models of a same software system consistent is a crucial requirement in model-driven software development to avoid inconsistencies and unexpected behaviours. In this section, we lay the foundations of consistency preservation and its variant when there are more than two metamodels, multi-model consistency preservation.

2.4.1. Consistency Relations

Models of a same software system share overlapping information about it. Therefore, they are said to be interrelated. For example, the object-oriented structure of a system can be represented in many different models such as its source code, UML class diagrams, communication diagrams, etc. Models are primary artefacts in model-driven software development. They are frequently and independently updated and their evolution defines the software system.

Modifying a model leads to *inconsistencies*. If a class of an object-oriented architecture is removed in the code, then it should also be removed in all other models where it appears. Otherwise, models are no more consistent. To avoid inconsistencies, it is possible to define relations between metamodels. These relations are called *consistency relations*. For example, a consistency relation can ensure that each class in the code of the software corresponds to a class in its class diagram and vice versa. Consistency relations are defined between metamodels and should hold between models that instantiate these metamodels.

Inconsistencies are most often a problem for developers, as they can lead to unexpected behaviours in implementations. Solving inconsistencies by defining consistency relations between metamodels is known as *consistency preservation*. One purpose of model-driven engineering being automation [HID+13], research mainly focuses on consistency preservation through automated or semi-automated mechanisms.

2.4.2. Model Transformations for Consistency Relations

2.4.2.1. Consistency Checking and Enforcement

There are multiple approaches for defining mechanisms to preserve consistency in a set of models. A common denominator in most of these approaches is the use of model transformations. Consistency preservation is a two-step process:

1. *Consistency checking*. Given a set of models, constraints in consistency relations are checked to ensure that models are consistent with each other.

2. *Consistency enforcement.* Given a set of models, if a constraint induced by a consistency relation is not fulfilled, models related by the constraint are updated until they are consistent again.

Enforcing consistency through the update of models can actually be performed by model transformations. For example, the QVT-R transformation language uses two execution modes: `checkonly` (consistency checking) and `enforce` (consistency enforcement). Transformations are either *state-based*, i.e. they are executed by comparing model states, or *delta-based*, i.e. they are represented as sequences of changes in models [Dis+11].

Consequently, a consistency relation can be associated with one or more model transformations. The relation provides the information to check consistency, whereas the transformation provides the information to restore it. Depending on the transformation language, one aspect may be more or less induced by the other. For example, transformation rules required to restore consistency are sometimes inferred from consistency relations.

2.4.2.2. Bidirectional and Incremental Transformations

As described in Section 2.1.3, there are several types of transformations. A useful category of transformations for preserving consistency is *bidirectional transformations* (bx) [Abo+18]; [Ste17a]. The purpose of bidirectionality in the context of consistency preservation is to avoid the duplication of transformations. All models related by a bx can act as source or targets models during the application of the bx. In other words, a single specification is enough to enforce consistency between multiple models in all directions.

Moreover, transformations considered for consistency preservation can be *incremental* [HLR06]. Restoring consistency usually involves updating a small part of models rather than generating them over again. Incrementality consists in propagating modifications only, so that parts of target models that are not covered by the transformation remain unaltered. Bidirectional and incremental transformations form an interesting category of model transformations for enforcing consistency.

2.4.2.3. Consistency Preservation Processes

Transformations and consistency relations form building blocks of approaches for preserving consistency. Then, it is possible to integrate them into general approaches for model-driven software development. The Vitruvius framework[4] is an approach for *view-centric* model-driven software development [KBL13]. It is built around the concepts of architecture *views* and *viewpoints* [ISO11]. Views are specific models whose role is to show particular elements of other models. Viewpoints are specifications of conventions for the construction, interpretation and use of views. Intuitively, views are to viewpoints as programs are to programming languages.

There is also a need for consistency preservation in Vitruvius. To this purpose, the framework executes consistency preservation transformations defined by means of two

[4] `http://vitruv.tools`

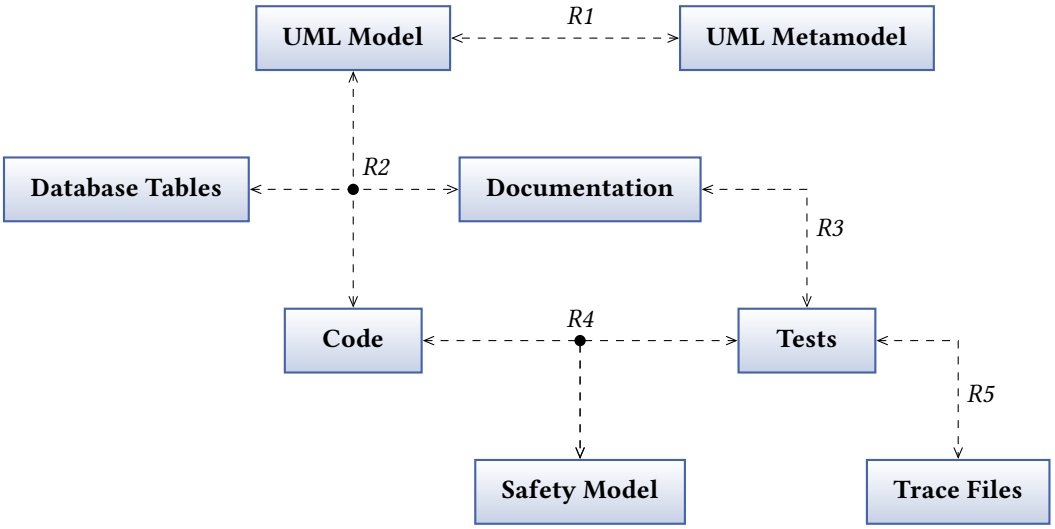


Figure 2.4.: Example of a consistency network with multiary relations

languages [Kra17]: the Mappings language (bidirectional, declarative) and the Reactions language (unidirectional, imperative). Consequently, Vitruvius embeds a consistency preservation process based on consistency relations and model transformations.

2.4.3. Multi-Model Consistency Preservation

2.4.3.1. Networks of Consistency Relations

The study of bidirectional and incremental transformations is often restricted to *binary transformations*, i.e. transformations for pairs of models. However, consistency relations can hold for more than two metamodels. Consider Figure 2.4[5], derived from [Ste17b].

Figure 2.4 is an example of *network of consistency relations*, i.e. a set of models with consistency relations between them. The network is consistent if and only if each consistency relation in it is fulfilled.

Consistency relations of the example can be (informally) defined as follows. *R1* ensures that the UML model conforms to the UML metamodel. *R2* specifies that code and database tables should be consistent with UML models as they are described and explained in the documentation. *R3* specifies that tests should reflect the required level of testing in the documentation. *R4* ensures that code and tests should be synchronized and adapted to the safety model. The safety model may assert its own conditions for test validation, e.g. a higher code coverage. *R5* specifies that trace files should contain information generated by the execution of tests.

Consistency relations (and bidirectional transformations associated with them) are said to be *multiary* when they relate more than two models. In Figure 2.4, relations *R2* and *R4* are multiary relations (respectively 4-ary and 3-ary relations).

An interesting result proved in [Ste17a], is that multiary bidirectional transformations may be defined in terms of binary bidirectional transformations under reasonable assump-

[5]This is a megamodel, i.e. a model of a system of models. [FN05]

tions. More precisely, it is not always possible to express a multiary bx as a set of binary bx (see WG1 in [Cle+19]). However, various approaches exist, e.g. by adding extra-models, so that in practice, it is reasonable to focus on binary bidirectional transformations. A strong benefit of this result is that bidirectional transformations can be independently developed for pairs of models and then combined. This approach is better suited to transformation developers and domain experts who rarely master all dependencies between models of the software system. Instead, they are specialized in the sense that they specify consistency relations for a small set of metamodels.

2.4.3.2. Interoperability Issues

Nevertheless, replacing multiary bidirectional transformations with sets of binary bidirectional transformations leads to problems. As stated in [Kla18], executing transformations one after the other, i.e. transitively, can create undefined behaviours that would not appear if each transformation was executed on its own. This problem is known as an *interoperability issue*.

Consider the relation *R2* of Figure 2.4. Suppose that this 4-ary bidirectional transformation is defined in terms of six binary bidirectional transformations. Say that the *Database Tables* model is updated and that a table is deleted. In the case of a 4-ary transformation, the *Database Tables* model becomes the source model during the execution of the transformation. Related elements in other metamodels, e.g. a class in *Code*, a class in a diagram of *UML Model* and a description of the class in the *Documentation*, are also deleted by executing the transformation.

In the case of multiple binary transformations, the update is propagated through the consistency network by applying transformations transitively. Therefore, by first applying the transformation *Database Tables* ↔ *UML Model*, the class is deleted in *UML Model*. Then, by applying *UML Model* ↔ *Code*, the class is deleted in the code. Third, by applying *Database Tables* ↔ *Code*, the transformation attempts to delete the class in the code. However, the class was already deleted in the second transformation. This is an example of interoperability issue resulting in an unexpected behaviour. It should be noted, though, that none of the transformations introduced an unexpected behaviour on its own.

Several approaches are proposed in [Kla18] to solve interoperability issues in multi-model consistency preservation. Such approaches aim to foster the independent development of binary transformations for consistency preservation. This thesis focuses on one of these approaches, the *decomposition of consistency relations*.

2.5. Constraint Satisfaction

Science and engineering branches often encounter problems characterized by a set of arbitrary variables (or objects) and a set of constraints that limit the values that variables can take. These problems are known as *constraint satisfaction problems*.

Computer science is no exception. For example, *task scheduling* is a problem in which a set of tasks should be processed by a set of computing resources such as CPUs under some goals such as minimizing response time [LL73]. Tasks and computing resources can be

regarded as *variables*, whereas problem requirements can be encoded as *constraints*, e.g. one task per resource at a time, all tasks should be processed once, etc.

2.5.1. Constraint Networks

The central concept of constraint satisfaction is the *constraint network*, a formalism encoding all the information of a constraint satisfaction problem [DC+03].

Definition 2.5.1 *A* **constraint network** $\mathcal{R}$ *is a tuple* (X, D, C) *such that* $X = \{x_1, \ldots, x_n\}$ *is a finite set of variables,* $D = \{D_1, \ldots, D_n\}$ *is a finite set of domains and* $C = \{C_1, \ldots, C_t\}$ *is a set of constraints. A domain* $D_i = \{v_1, \ldots, v_k\}$ *lists the possible values of the variable* x_i*. A constraint* C_i *is a relation* R_i *defined on a subset of variables* S_i*, i.e.* $S_i \subseteq X$*. As a result, if* $S_i = \{x_{i_1}, \ldots, x_{i_r}\}$*, then* $R_i \subseteq D_{i_1} \times \cdots \times D_{i_r}$*.*

In other words, a constraint C_i is a couple (S_i, R_i) with a subset of variables S_i and a relation R_i that is made up of all simultaneous value assignments that satisfy the constraint. The *arity* of a constraint C_i is equal to the cardinality of its set of variables S_i. For example, a binary constraint relates exactly two variables.

Definition 2.5.2 *An* **instantiation** *of a subset of variables* S *of a constraint network* $\mathcal{R} = (X, D, C)$ *is an assignment of variables in* S*. For* $S = \{x_{i_1}, \ldots, x_{i_k}\}$*, an instantiation of* S *can be regarded as a tuple of ordered pairs* $\{(x_{i_1}, a_{i_1}), \ldots, (x_{i_k}, a_{i_k})\}$ *where* a_{i_j} *a value of the domain* D_{i_j} *for the variable* x_{i_j}*.*

Definition 2.5.3 *A* **solution** *of a constraint network* $\mathcal{R} = (X, D, C)$ *is an instantatiation of* X*, i.e. of all variables of the network, such that each constraint* C_i *is satisfied. A constraint* $C_i = (S_i, R_i)$ *is satisfied if the tuple made up of the assignment of each variable in* S_i *subset of variables belongs to* R_i*.*

In other words, an instantiation assigns to some variables a value of their respective domains. A solution is an instantiation of all variables of the network such that values of variables satisfy all constraints of the network.

2.5.2. Constraint Graphs and Hypergraphs

If all constraints are binary, it is possible to represent the network by a graph. An edge between two variables indicate that there exists a constraint between them.

Definition 2.5.4 *A constraint network* $\mathcal{R} = (X, D, C)$ *whose all constraints are binary can be represented by a* **constraint graph** $\mathcal{G}_\mathcal{R} = (V_\mathcal{R}, E_\mathcal{R})$ *where* $V_\mathcal{R} = X$ *and* $E_\mathcal{R}$ *is the set of edges that link all vertices of a same constraint, i.e.* $E_\mathcal{R} = \{\{x_{i_1}, x_{i_2}\} \mid \forall\, C_i = (\{x_{i_1}, x_{i_2}\}, R_i) \in C\}$*.*

In practice, many constraint satisfaction problems can be modeled with binary constraints and represented by constraint graphs. However, it is sometimes useful to visualize any type of constraint network with a more general formalism. For this reason, we define hypergraphs and constraint hypergraphs.

Definition 2.5.5 *A* **hypergraph** *is a couple* $\mathcal{H} = (V, E)$ *where* V *is a set of vertices and* E *is a set of hyperedges, i.e. non-empty subsets of* V*. Thus,* $E \subseteq \mathcal{P}(V) \setminus \{\varnothing\}$*.*

Definition 2.5.6 *A constraint network* $\mathcal{R} = (X, D, C)$ *can be represented by a* **constraint hypergraph** $\mathcal{H}_{\mathcal{R}} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ *where* $V_{\mathcal{R}} = X$ *and* $E_{\mathcal{R}} = \{S_i \mid \forall\, C_i = (S_i, R_i) \in C\}$*.*

Constraints graphs and hypergraphs make it easier to find solutions of constraint satisfaction problems (CSPs). Classic algorithms for finding solutions to CSPs include constraint propagation, backtracking, backjumping, etc [DF02]. In the context of this thesis, we are mainly interested in the formalization of CSPs in order to adapt them to the problem of multi-model consistency preservation. As a result, we will design our own algorithm to perform a decomposition of consistency specifications.

2.6. Automated Deduction

This section presents *automated deduction*, a subset of automated reasoning that focuses on the automated production of proofs. Automated deduction tools, also called automated theorem provers, are widely used nowadays in software engineering. They are part of *formal methods*, a set of techniques that aim to develop more reliable software.

Formal methods can, for example, be used in order to show the absence of bugs in a program whereas testing can only show their presence. Accordingly, they participate in the development of software that is verified (i.e. that respects the specifications) and validated (i.e. that meets user's needs). In model-driven software development, formal methods can be used to analyze and prove properties on models and model transformations.

First, we briefly introduce first-order logic, a formal language for automated reasoning, and its main properties regarding automated deduction. Second, we present the Satisfiability Modulo Theories (SMT) problem, a decision problem using first-order logic and its relationship with automated theorem proving.

2.6.1. First-Order Logic

Formal methods use formal languages to reason about programs. Logics are an appropriate choice for this purpose: they have strong theoretical foundations and they are abstract enough for modeling many problems and reasoning about them. There are many types of logics representing various languages for various needs. For example, temporal logics are well suited for model checking techniques [Alu+95], whereas fuzzy logics are applied to problems without sharp boundaries [Sin+13].

2.6.1.1. Syntax of First-Order Logic

First-order logic is another language for formalizing of mathematics [Smu12]. It can be regarded as an extension of propositional logic with quantifiers and variables. The syntax of propositional logic is made up of propositions that are either TRUE or FALSE and logical connectives ($\wedge$, $\vee$, $\neg$, etc.). Propositional logic is generally too limited for formal methods.

For example, we need to be able to model the fact that a property must hold *whatever* the value of a given variable in a program.

The solution is to extend propositional logic. First, we introduce terms, i.e. variables, e.g. x, and functions of these variables, e.g. $f(x, y)$ or $f(f(x))$. Second, we extend propositions, e.g. p, with predicates, e.g. $p(x)$. Predicates still evaluate either to TRUE or FALSE but they can also include zero or more arguments. A predicate with zero argument is a proposition. Therefore, first-order logic is much more expressive than propositional logic. For example, a statement like "$\sqrt{x} > 0$" can be written with one variable (x), one function ($\sqrt{.}$) and one predicate (`isPositive`). In addition to logical connectives of propositional logic, first-order logic also comes with an existential quantifier ($\exists$) and an universal quantifier ($\forall$). Quantifiers are useful to reason about the quantity of variables satisfying a formula.

Functions and quantifiers make finite first-order formulae more expressive than finite propositional formulae. Regarding consistency specifications, it facilitates the modeling of relations between metamodel elements that form consistency rules.

2.6.1.2. Semantics of First-Order Logic

The other important aspect of first-order logic after syntax is semantics. Once a specification has been modeled with first-order logic, i.e. the input language of automated theorem provers, we aim to evaluate the resulting formula by assigning a meaning to predicate symbols, constant symbols and function symbols. Such an assignment is called an *interpretation*. Two important concepts follow the notion of interpretation:

- *Satisfiability*. A formula is *satisfiable* if there exists an interpretation making it true;
- *Validity*. A formula is *valid* if it is true under every interpretation.

Theorem proving can be regarded as a validity problem. The theorem can be encoded as a first-order formula. Then, it is proved if every interpretation makes the theorem evaluate to TRUE. In this thesis, we show how the removal of a consistency relation can be encoded as a first-order formula and compared with theorem proving. Proving that a formula is true under *every* interpretation is hard. Instead, a common approach is to check the unsatisfiability of the negated formula.

A formula F and its negation $\neg F$ are contradictory. If F evaluates to TRUE under a given interpretation, $\neg F$ evaluates to FALSE and vice versa. Consequently, F is valid if and only if $\neg F$ is unsatisfiable. For this reason, automated theorem provers are mostly SAT solvers, i.e. programs that take a formula as an input and answer whether it is satisfiable.

Satisfiability in first-order logic is undecidable [Chu36]. That is, there is no algorithm that takes a first-order formula as an input and always tell if it is satisfiable. As a result, automated theorem proving is not always possible. Even for propositional formulae, it is believed that there are only exponential time algorithms, e.g. DPLL. However, current SAT solvers are heavily optimized and remain effective for most practical cases.

2.6.2. Satisfiability Modulo Theories

Modeling theorems, programs or consistency rules with first-order logic is hard. They are composed of arithmetic operations, strings, arrays, bit vectors, etc. A variant of satisfiability is *satisfiability modulo theories* (SMT). SMT instances are formulae based on theories, i.e. logical axioms and consequences of these axioms. For example, the following formula is a valid SMT instance based on a theory for arithmetic.

$$(a = b \times 2) \wedge (b = c \times 3) \implies (a = c \times 6)$$

SMT instances have their own solvers (called SMT solvers, such as Z3 [DB08]) based on algorithms such as DPLL(T), a variant of DPLL with theories.

3. Consistency Preservation

Consistency preservation in model-driven software development can be achieved through the explicit definition of consistency rules [Kra17]. Rules are represented as relations between metamodels. As a result, specifying consistency rules for a large number of metamodels can involve a large number of consistency relations and a significant computational cost for consistency preservation algorithms.

Following the idea that problems are easier to solve when they are divided into multiple subproblems, this problem can be partially solved by applying consistency preservation on small subsets of metamodels, later combined to ensure a global consistency.

Given a set of metamodels, decomposition of consistency relations is a procedure to transform a set of consistency rules into independent, smaller subsets of consistency rules while ensuring that the consistency specification remains unaltered.

In this context, this chapter formalizes the concept of consistency as used in the decomposition procedure and shows how the QVT-R transformation language can be used to achieve consistency in practice.

3.1. Description of Consistency Relations

3.1.1. Consistency Relation Graph

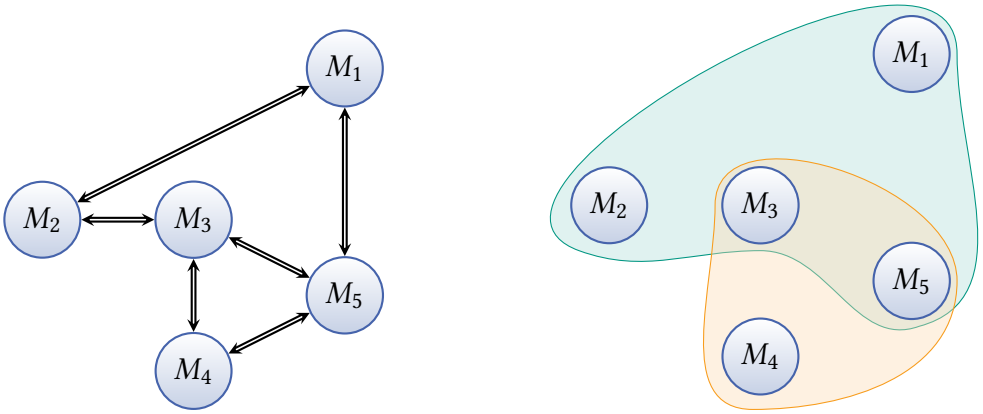
Consistency is general property of models whose specification depends on a set of metamodels. Not all metamodels are necessarily interrelated. Therefore, the first step of specifying consistency is to describe which metamodels own redundant information.

Objects and relations between them can be modeled with graphs. In the context of consistency preservation, such a graph is called a consistency relation graph.

Definition 3.1.1 *A* **consistency relation graph** *is a graph* $\mathcal{G} = (\mathcal{M}, C)$ *where* $\mathcal{M}$ *is a set of metamodels and* C *is a set of consistency relations on* $\mathcal{M}$*, i.e.* $C \subseteq \mathcal{M} \times \mathcal{M}$*.*

This definition is restrictive: consistency relations are binary relations only. However, it is possible for n metamodels to be all interrelated with each other. Such as case would be preferably modeled with n-ary relations and hypergraphs.

In a hypergraph, the set of consistency relations C is a subset of the power set of $\mathcal{M}$, i.e. $C \subseteq \mathcal{P}(\mathcal{M}) \setminus (\{\varnothing\} \cup \bigcup_{M \in \mathcal{M}} \{M\})$. This way, any subset of the set of metamodels of size at least two can form a consistency relation. Choosing consistency relation *graphs* over *hypergraphs* is motivated by the fact that consistency relation graphs only describe one aspect of consistency. Other important aspects include consistency between metamodel elements and consistency repair. The former aspect indicates which parts of metamodels



(a) Consistency relation graph

(b) Consistency relation hypergraph

Figure 3.1.: Two possible representations of a consistency network

represent redundant information and are concerned for this reason by needs for consistency. The latter aspect involves model transformations to enforce consistency if it was destroyed.

Some categories of model transformations are well suited to consistency repair, such as bidirectional transformations. An important result about bidirectional transformations (see Section 2.4.3) is that n-ary bidirectional transformations can be reasonably expressed in practice as sets of binary bidirectional transformations.

By favouring the development of independent binary consistency relations, this approach also increases modularity in the specification. Under this approach, the update of a metamodel only requires the update of consistency relations in which the metamodel is involved. This has no influence on other consistency relations and metamodels. On the contrary, a consistency relation hypergraph may allow the definition of a single, global consistency relation with all metamodels involved. In that case, it is impossible to check that instances of a subset of metamodels are consistent with each other without checking the consistency of instances of the whole set of metamodels.

3.1.2. Consistency Rule

An edge between two metamodels in the consistency relation graph (i.e. a consistency relation) means that their instances must be consistent with each other. A *consistency rule* defines the conditions under which they are. The following definitions are adapted from the theoretical framework of [Kra17] to describe metamodeling and consistency preservation. See Section 2.3 for a definition of metamodels and instances.

Definition 3.1.2 *Let $M \in \mathcal{M}$ be a metamodel and let $\mathbb{c} = \langle \mathbb{c}_1, \ldots, \mathbb{c}_n \rangle$ be a metaclass tuple of M. A* **condition** *for $\mathbb{c}$, denoted $COND_{\langle \mathbb{c}_i \rangle}$, is a subset of all instances of $\mathbb{c}$, i.e. $COND_{\langle \mathbb{c}_i \rangle} \subseteq \mathcal{I}(\mathbb{c}_1) \times \ldots \times \mathcal{I}(\mathbb{c}_n)$.*

Definition 3.1.3 *Let M_l and $M_r \in \mathcal{M}$ be two metamodels and let $\mathbb{c}_l = \langle \mathbb{c}_{l_1}, \ldots, \mathbb{c}_{l_n} \rangle$ and $\mathbb{c}_r = \langle \mathbb{c}_{r_1}, \ldots, \mathbb{c}_{r_n} \rangle$ be metaclass tuples of M_l and M_r respectively. A* **consistency rule** *is a set $\mathcal{R}_{\mathbb{c}_l, \mathbb{c}_r} \subseteq COND_{\langle \mathbb{c}_l \rangle} \times COND_{\langle \mathbb{c}_r \rangle}$, i.e. a set of pairs of co-occurring instance tuples that satisfy $COND_{\langle \mathbb{c}_l \rangle}$ and $COND_{\langle \mathbb{c}_r \rangle}$ respectively.*

Given two metamodels M_l and M_r, not all instances of M_l are consistent with instances of M_r and vice versa. The main idea behind consistency rules is to build a set that contains only consistent pairs of instances of metaclass tuples. Instances of metaclass tuples are made of objects, i.e. model elements.

A consistency rule is built in two steps. First, conditions are defined: they act as filters for a given metamodel. Metaclasses are selected to form a metaclass tuple. Metaclass tuples are chosen over the whole metamodel to identify elements of the metamodel that contribute to consistency preservation. For example, if consistency depends only on the value of an attribute of a metaclass, only this metaclass will be included in the metaclass tuple as other metaclasses do not alter consistency. Among all instances of metaclass tuples (i.e. tuples of objects), the COND set contains those who fulfill the condition.

Secondly, the consistency rule *binds* its two conditions. In other words, the rule selects pairs of tuples of objects according to the following principle: if one tuple of objects occurs in a model, then the other tuple of objects must occur in the other model.

The definition of *consistency rule fullfilment* formalizes this principle.

Definition 3.1.4 *Let $I_{M_1} \in \mathcal{I}(M_1)$ and $I_{M_2} \in \mathcal{I}(M_2)$ be two instances of two distinct metamodels M_1 and M_2. The unordered pair $\underline{I} = \{I_{M_1}, I_{M_2}\}$* **fulfills** *a consistency rule $\mathcal{R}$ if:*

$$\forall \langle cr_l, cr_r \rangle \in \mathcal{R},\ (\exists\, I_{M_l} \in \underline{I} : cr_l \subseteq I_{M_l} \iff \exists\, I_{M_r} \in \underline{I} : cr_r \subseteq I_{M_r})$$

Note that a consistency rule is still fulfilled if no tuple of objects occurs in metamodel instances. In other words, consistency is destroyed if and only if, for a given pair of tuples of objects in a consistency rule, the left (resp. right) metamodel instance contains the left (resp. right) tuple of object whereas the right (resp. left) metamodel instance does not contain the right (resp. left) tuple of object.

Example 3.1.1 *Let* Person *be a metaclass of a metamodel* P *and let* Employee *be a metaclass of a metamodel* E. *Let $\mathcal{R}$ be a consistency rule that asserts that each* name *attribute of* Person *objects must correspond to a* name *attribute in* Employee *objects and vice-versa.*

Table 3.1 summarizes the cases in which instances of P *and* E *are consistent.*

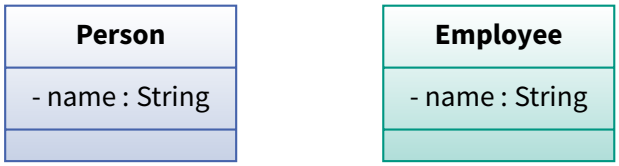


Figure 3.2.: Two metamodels, each with one metaclass

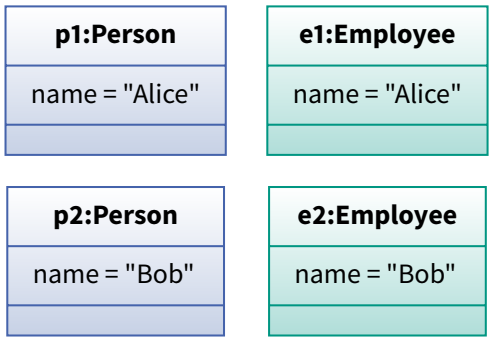
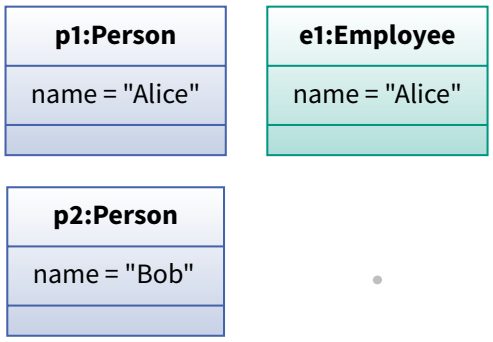
Model 1	Model 2	Consistency
		*The metamodel instances are consistent. Each* Person *object in the first model can be related to an* Employee *object in the second model so that their attributes have the same value, and vice-versa.*
		*The metamodel instances are not consistent because there is no* Employee *object whose* name *attribute has the value* "Bob"*. As a result, the consistency rule is not fulfilled by these instances.*
		The metamodel instances are consistent. They are empty. Therefore, there is no object that does not match an object of the other model. The consistency rule is fulfilled.

Table 3.1.: Consistency of a few instances of Figure 3.2

3.1.3. Consistency Specification

Specifying consistency in a set of metamodels can be achieved by defining as many consistency rules between pairs of metamodels as necessary.

Definition 3.1.5 *A* **consistency specification** *for a set $\mathcal{M}$ of metamodels is a set $\underline{\mathcal{R}}$ of consistency rules for elements of $\mathcal{M}$.*

Consequently, checking that all models in a set of models are consistent with each other can be achieved by checking that there exists no pair of models (i.e. metamodel instances) that do not fulfill a consistency rule.

Definition 3.1.6 *Let $\mathcal{I}$ be a set of metamodel instances and let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$. $\mathcal{I}$ is* **consistent** *if and only if:*

$$\forall\, \mathcal{R} \in \underline{\mathcal{R}},\ \forall\, (I_l, I_r) \in \mathcal{I} \times \mathcal{I} : \{I_l, I_r\} \textit{ fulfills } \mathcal{R}$$

It is possible to define a consistency specification without consistency relation graph. The reason for this is that the consistency relation graph can be built from consistency rules. Given a set $\mathcal{M}$ of metamodels and a set $\underline{\mathcal{R}}$ of consistency rules on $\mathcal{M}$, a consistency relation graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ can be defined as follows:

- $V_\mathcal{G} = \mathcal{M}$
- $E_\mathcal{G} = \{\{M_l, M_r\} \mid \exists\, \mathbb{c}_l \in M_l, \exists\, \mathbb{c}_r \in M_r : \mathcal{R}_{\mathbb{c}_l, \mathbb{c}_r} \in \underline{\mathcal{R}}\}$

The consistency relation graph remains an important structure for the decomposition procedure. For example, it can be used to identify isolated metamodels or independent subsets of metamodels without referring to conditions, metaclass tuples and instances. For this reason, it may be convenient to consider the consistency relation graph as a fledged object of the consistency specification. In doing so, it is necessary to check that one consistency relation is always associated with one or more consistency rules.

3.2. Consistency with QVT-R

The last important point to define a complete, usable consistency specification is to define how conditions are chosen and how consistency rules bind these conditions. Conditions being possibly infinite sets of objects, intensional definitions of conditions are the most appropriate way to filter instances in the construction of consistency rules. This section shows the interest of QVT-R as a language to specify consistency.

The *QVT Relations* language (abbreviated as QVT-R) is a declarative transformation language. It is part of QVT (*Query/View/Transformation*), a standard to specify model transformations and consistency. See Section 2.2.3 for an introduction to QVT.

A QVT-R specification is made of *relations* which represent relationships between metamodel elements. The declarative nature of QVT-R implies that it specifies *what* objects need to be consistent rather than *how* consistency should be ensured or restored.

This task is delegated to *QVT Core* (QVT-C), another subset of QVT, whose functioning is imperative. The transition from QVT-R to QVT-C is achieved by a *RelationsToCore* transformation. Since many concepts are common to QVT-R and QVT-C, the standard provides a common structure called *QVT Base*.

When a set of models is provided, along with a metamodel, the QVT-R specification can be executed. More precisely, relations in QVT-R can be run in two modes: `checkonly` and `enforce`. The former checks that models fulfill the consistency specification. The latter is able to perform modifications on target models so that consistency is restored.

The aim of the decomposition procedure is to make changes to the consistency specification itself in order to generate a new specification. Such an operation requires a static analysis of QVT-R files and metamodels.

For this reason, QVT-R specifications do not need to be runned. Moreover, metamodel instances are not involved in the decomposition procedure.

3.3. Structure of a QVT-R Specification

A consistency specification using QVT-R consists of several QVT-R files. Defining one binary consistency relation per file is a good practice. It fosters separation of concerns and relations can be developed independently. Each file is composed of at least one transformation, each transformation itself being composed of at least one relation.

Listing 3.1 illustrates the use of QVT-R to formalize consistency of Example 3.1.1.

```
1  import perspack : 'person.ecore'::Person;
2  import emppack : 'employee.ecore'::Employee;
3
4  transformation persEmp(pers:perspack, emp:emppack) {
5
6      top relation PersonEmployee {
7          n: String;
8
9          domain pers p:Person {name=n};
10         domain emp e:Employee {name=n};
11     }
12 }
```

Listing 3.1: Consistency specification of Example 3.1.1 with QVT-R

The most important concepts of the QVT-R language are explained below. An extensive specification of the language has been drawn up by the *Object Management Group* (OMG) as part of the *Meta-Object Facility* (MOF) standard [Obj16b].

3.3.1. Imports

Imports are statements to make metamodels available for transformations. Metamodels are referenced through URI (e.g. `'person.ecore'`). In the context of this work, URI will always refer to Ecore files. An import means that the root element of the metamodel (most often, a *package*) can be resolved in transformations.

It is also possible to use an alias, such as `perspack` in the example, to resolve the root element. Given a URI, the `::` syntax allows more specific imports by retrieving a direct child of the source element (e.g. a subpackage). This is also optional. Imports are introduced with the keyword **`import`**.

3.3.2. Relational Transformations

Transformations are the most general objects of QVT. There are several types of transformations, including *relational* transformations for QVT-R. Their purpose is to group relations in order to formalize consistency between metamodels. Relational transformations gain access to metamodels via *model parameters* (e.g. `pers` and `emp` in Creflst:Overview:ConsistencyRelations:ExampleQVTR).

There is at least one model parameter per transformation. In this work, each transformation will have exactly two parameters in order to match the definition of a consistency relation graph. Relational transformations can also include keys, i.e. sets of properties to guide the creation or the update of objects in case of consistency enforcement.

Given that the decomposition procedure does not address metamodel instances, the use of keys will be ignored. Relational transformations are introduced with the keyword **`transformation`**.

3.3.3. Relations

Relations are part of relational transformations. They can be seen as sets of constraints on metamodels. More precisely, they express constraints (on metamodel elements) that metamodel instances must fulfill in order to be consistent. A relation is able to use elements of metamodels passed as parameters of the transformation to which it belongs.

Relations use *pattern matching* to bind elements from different metamodels. To this purpose, relations define variables, such as `n : String` in the example. Instances must conform to the pattern created by the variables. For example, the `name` attributes of both metaclasses in Listing 3.1 are bound to the same variable `n`. Therefore, values of their instances must be equal because they match the same pattern.

Relations offer some additional features. First, there is a hierarchy of relations in a transformation. Some relations are said to be *top-level*. In terms of QVT-R syntax, these relations are prefixed with the **`top`** keyword. When a transformation is executed, only top-level relations will be invoked. Non-top-level relations can be invoked from other relations thanks to the **`where`** keyword. Also, a relation can contain a **`where`** clause and a **`when`** clause. These clauses act as additional constraints in the relation.

The **`where`** clause acts as an *invariant* for the relation. More precisely, the condition expressed by the **`where`** clause must be fulfilled by all model elements of the relation and

at any time during its processing. A condition can contain two types of expressions: usual OCL expressions (with variables and metamodel elements) or invocations of non-top-level-relations. An invocation consists of the name of the relation to invoke and variables of the current relation that the invoked relation will be able to use.

The `when` clause acts as a *precondition* for the relation. Therefore, the condition expressed by the `when` clause is the first thing evaluated during the execution of a relation. If the condition is not fulfilled, then the relation is not executed. A condition can contain usual OCL expressions. However, unlike `where` conditions, only top-level-relation invocations are possible in `when` clauses.

```
1  transformation persEmp(pers:perspack, emp:emppack) {
2
3      top relation PersonEmployee {
4          n: String;
5
6          domain pers p:Person {name=n};
7          domain emp e:Employee {name=n};
8
9          where {PersonEmployeeAddress(p, e);}
10     }
11
12     relation PersonEmployeeAddress {
13         ...
14     }
15 }
```

Listing 3.2: Extension of Listing 3.1 with a `where` clause

Example 3.3.1 *Suppose that the example of Example 3.1.1 is extended so that both metamodels now contain an* Address *metaclass and that the* Person *and* Employee *contain a reference to the* Address *metaclass. The new consistency specification requires that a* Person *and an* Employee *having the same name should have the same address.*

As shown in Listing 3.2, this can be achieved by adding a new non-top-level relation in the transformation, which is only invoked from the existing `where` *clause. The new* PersonEmployeeAddress *relation can use variables* `p` *and* `e` *with the guarantee that they represent consistent objects according to the* PersonEmployee *relation.*

Invariants and preconditions are a way of composing relations in QVT-R and thus building complex relations from simple relations. Finally, relations include domains to access metamodel elements. Relations are introduced with the keyword `relation`.

3.3.4. Relation Domains

Domains are a QVT concept to describe under which conditions a model is conform. There are several types of domains, relation domains being those used in QVT-R. This description of "conforming models" is achieved by the definition of *domain patterns*.

First, a domain assigns metamodel elements to variables. These variables are called the root variables of the domain. Elements should all be part of the same metamodel passed as a parameter to the parent transformation.

Then, the set of admissible instances for a given element is restricted by defining a pattern. This restriction is achieved by limiting the values that *properties* of the element instances can take. For example, properties of a metaclass are its attributes and everything resolvable by its references. A pattern, such as `p:Person {name=n}`, is made up of a variable (`p`) that refers to a metaclass (`Person`), and a *template expression* (`{name = n}`), i.e. a template that objects must match to be considered valid.

There are two types of template expressions: *object* template expressions in which the root variable is assigned to a single element and *collection* template expressions which match collections of model elements. An object template expression includes a (possibly empty) list of *property template items* (PTIs), i.e. assignments of properties to values. In Example 3.3.1, there are two domains, both with one pattern (that is, one template expression). The domain pattern of `pers` (the *Person* metamodel) is composed of one property template item, `name = n`.

Morever, a template expression can include its own invariant (written between braces after the set of PTIs). This invariant can be seen as a local **`where`** clause for the pattern.

Regarding the execution of QVT-R, domains are where running modes are chosen. If a domain is not prefixed or prefixed with **`checkonly`**, instances of its metamodel element will not be updated. If a domain is prefixed with **`enforce`**, instances (called "target models") will be updated to enforce consistency.

There is another use of domains in QVT-R: *primitive* domains. The **`primitive`** keyword indicates that the domain has no pattern. It consists only of a variable declaration, whose type is a primitive type (`Integer`, `String`, ...). This structure is useful to pass information such as constants between relations. The value of the constant, passed as a parameter of a relation invokation, will be captured by the primitive domain and can therefore be used in other (relation) domains.

A relation includes at least two domains. In the context of binary consistency relations, relations supported by the decomposition procedure will include exactly two domains. Relation domains are introduced with they keyword **`domain`**.

3.3.5. Expressions and Conditions

Property template items, invariants (**`where`**) and preconditions (**`when`**) are the QVT-R structures in which conditions on metamodel elements occur. These conditions are also called *expressions*. For example, in Listing 3.2, there are three OCL expressions: "`name = n`" twice and one *relation call expression*, "`PersonEmployeeAddress(p, e);`".

The language used in QVT-R to express conditions is *EssentialOCL*, an adapted version of OCL for declarative, relational transformation languages.

Although most often used to specify static semantics of metamodels, OCL allows the specification of complex constraints between QVT-R variables and metamodel elements (e.g. arithmetic operations, string manipulation, operations on sets of elements, etc.). This use imposes some restrictions on OCL, hence the creation of EssentialOCL.

QVT-R concept	Consistency concept
Set of transformations	Consistency specification
Transformation	Consistency rule
Relation	Part of a consistency rule
Domain	Condition on a metaclass tuple
Domain pattern (template expression)	Condition on a metaclass
Property template item	Part of a condition on a metaclass

Table 3.2.: Comparable concepts in QVT-R and consistency definitions

For example, OCL allows the validation of pre- and postconditions of operations using special keywords inside expressions. In these conditions, it is especially possible to compare the value of an object's property with the value it had before. This feature implies the existence of an internal state machine to handle object mutation. Moreover, objects undergoing the validation of OCL constraints may be in an unstable state.

This is however incompatible with the declarative nature of QVT-R in which objects are stable, whether local or passed as parameters of other QVT-R relations. That is why this OCL feature does not exist in EssentialOCL. Consequently, it is side-effect free.

Nevertheless, preconditions and invariants can still be expressed in QVT-R. They are represented at the relation level and validated according to QVT-R semantics rather than OCL semantics. The benefit of this approach is that if a constraint is not validated, then the whole relation does not hold and there can be no unstable object in this relation. The decomposition procedure requires the static analysis of OCL expressions to compare consistency specifications. For a comprehensive explanation on the use of OCL in the decomposition procedure, refer to Chapter 6.

All the concepts mentioned above are summarized in Figure 3.3, which is a lightened view of concepts participating in QVT-R model transformations and occuring in different subsets of QVT (*Relations*, *Template*, *Core*, *Base*).

3.4. From QVT-R to Consistency Rules

The way to specify consistency using QVT-R is similar to the theoretical framework presented in Section 3.1. It is possible to establish a correspondence between these two approaches, so that QVT-R becomes a well-suited language for the problem of multi-model consistency preservation.

The main difference between the two approaches is that the definitions of the consistency framework only focus on which instances are valid, whereas the QVT-R language allows to explain how to choose them. Thus, a simple concept of the consistency framework is often split into several constructs in QVT-R. Specifying consistency for a set of metamodels is best achieved through a bottom-up approach. The first step is to identify, for each metamodel, which elements play a role in the definition of consistency in order to build *conditions*. The second step is to bind metamodels by selecting instances from the first

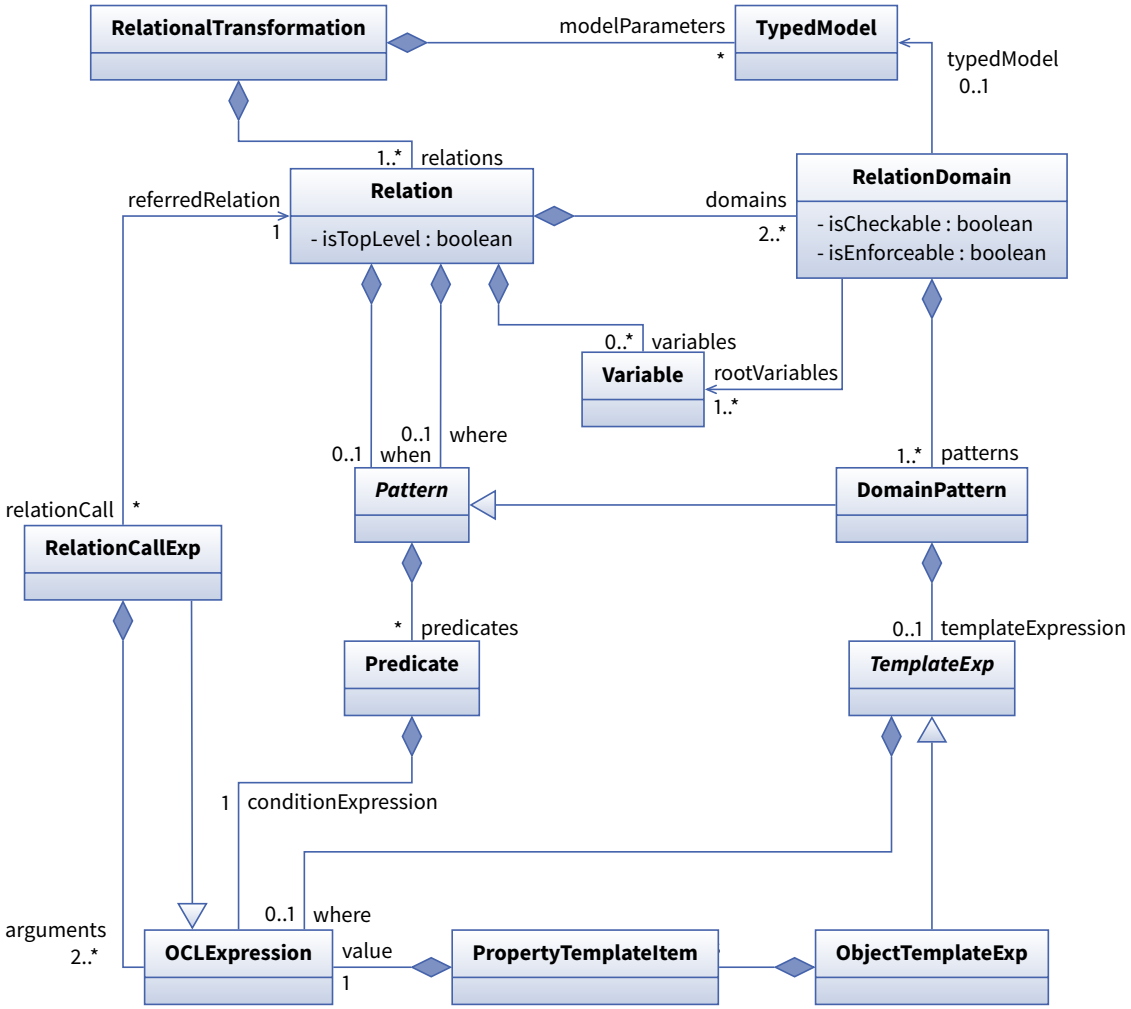


Figure 3.3.: Simplified class diagram of QVT-R concepts

condition that match instances from the second condition in order to build *consistency rules*. Repeating this for each pair of metamodels leads to a global *consistency specification*. This approach is adopted in this section to explain how QVT-R can be used as a means of multi-model consistency specification.

3.4.1. From Domain Pattern to Condition on a Metaclass

In QVT-R, a domain pattern can be regarded as a template for objects. More precisely, the domain pattern first declares a metaclass and then a template expression, made up of property template items. The latter act as criteria for instances. If a given instance of the metaclass matches these criteria, it is considered valid.

Listing 3.3 demonstrates three domain patterns based on Figure 3.2. The first pattern `p1` will filter every *Person* object whose `name` attribute has value `'Alice'`. The second pattern `p2` will filter every *Person* object. Since the variable `n1` is not bound, it can take any value (either `undefined`, empty or non-empty strings). Finally, the third pattern `p3` will filter every *Person* object whose `name` has the suffix `'␣Jr.'` because `n2` is not bound.

```
1 domain pers p1:Person {name = 'Alice'},
2             p2:Person {name = n1},
3             p3:Person {name = n2 + '␣Jr.'};
```

Listing 3.3: Three domain patterns

When an object template expression is made up of several property template items, all have to be fulfilled for the instance to be valid. Given that an object in the consistency framework is fully determined by its properties (attributes and references), QVT-R patterns provide a sufficiently powerful means of filtering objects for consistency preservation purposes. Consequently, a domain pattern is able to filter objects (i.e. metaclass instances) based on their properties. This matches the behavior of conditions. For a single metaclass $\mathbb{C}_1$, an object belongs to the condition $\text{COND}_{\langle \mathbb{C}_1 \rangle}$ if and only if it is a valid instance according to the domain pattern.

3.4.2. From Domain to Condition on a Metaclass Tuple

Since consistency specifications often involve more than one metaclass per metamodel, conditions are defined on metaclass tuples. The order of metaclasses in the tuple is arbitrary: it is a convenient way to select subsets of instances of metaclasses that are not necessarily distinct.

QVT-R offers a similar way of expressing conditions on metaclass tuples by allowing domains to have multiple domain patterns. For example, the three domain patterns of Listing 3.3 can be seen as a metaclass tuple (*Person*, *Person*, *Person*). According to the QVT standard, nothing prevents two domains from the same relation to refer to the same metamodel. Therefore, the construction of a unique metaclass tuple for a given metamodel can be achieved by concatenating the tuples of all domains that refer to this metamodel.

Regarding semantics of consistency checking in QVT-R, relations are always checked in a particular direction. That is, a typed model is chosen among all those in the domains of the relation. The verification then consists in ensuring that the instances of the chosen model fulfill the domain patterns, given instances of other models. Bidirectionality can be achieved by checking consistency for every typed model of the relation.

An important point is that domain patterns are checked separately. This results in the fact that a single object can fulfill multiple domain patterns. For example, let o_1 be an object *Person* whose *name* attribute is `'Alice'` and let o_2 be an object *Person* whose *name* attribute is `'Bob␣Jr.'`. This instance with two objects satisfies the three domain patterns of Listing 3.3. More precisely, o_1 fulfills `p1` and o_2 fulfills both `p2` and `p3`.

Another consequence of checking semantics in QVT-R is that empty instances do not destroy consistency. This is consistent with the definition of consistency rule fullfilment (Definition 3.1.4).

3.4.3. From Transformation to Consistency Rule

Given conditions on metaclass tuples $\langle \mathbb{c}_l \rangle$ and $\langle \mathbb{c}_r \rangle$, the last step to specify consistency between two metamodels is to select a subset of $\text{COND}_{\langle \mathbb{c}_l \rangle} \times \text{COND}_{\langle \mathbb{c}_r \rangle}$. This subset, called consistency rule, represents the set of consistent pairs of instances.

Relations in QVT-R promote modularity. A common design pattern for model transformations (known as *phased construction* [LK14]) consists in the creation of so-called *phases* that reproduce the hierarchical structure of metamodels. Insofar as possible, each relation binds one metaclass of the first metamodel with one metaclass of the second metamodel. Relating phases can be achieved with relation invokations (**`where`** and **`when`** clauses).

Consequently, metaclass tuples can be distributed over multiple relations. There again, metaclass tuples can be concatenated by retrieving domains that refer to the same metamodel in different relations. A simple way to do this is to consider a transformation as a tree where relations are nodes and relation invokations are directed edges. A breadth-first search starting from top-level relations on this graph allows to visit all relations in the right order and to gradually concatenate metaclass tuples for a given metamodel.

The interest of QVT-R relations lies in variables. When they are bound, variables restrict values that can be taken by objects of a metamodel depending on values of objects of the other metamodel. This is how variables bind elements from different metamodels.

Two domains are introduced in Listing 3.4. Taken separately, each domain defines (through the domain patterns) valid instances of the metamodel it refers to. The domain `emp` accepts all instances in which there is at least one *Employee* object. The domain `pers` accepts all instances in which there is at least two *Person* objects with the following additional constraint: not all *Person* objects must have the same *name* value. This constraint is a consequence of the local **`where`** clause of the template expression of `p2:Person`.

```
1  relation PersonEmployee {
2      n1: String;
3      n2: String;
4
5      domain emp e:Employee {name = n1};
6      domain pers p1:Person {name = n1},
7                  p2:Person {name = n2} {n1 <> n2};
8  }
```

Listing 3.4: Using QVT-R variables to model consistency rules

However, when these domains are combined, not all pairs of instances of `emp` and `pers` are valid. The relation requires that `e.name` and `p1.name` have the same value. In other words, the relation makes a selection among all pairs of instances of metaclass tuples so that variables have valid assignments. This mechanism is similar to the creation of a consistency rule in which pairs of instances are also filtered.

Variables in QVT-R can be used outside domain patterns: preconditions (**`when`**) and invariants (**`where`**) also include template expressions. Moreover, variables are not always local to a relation. In the case of a relation invokation, it is possible to export bound

variables via relation invokations. This is frequent in a phased construction of consistency specifications.

According to checking semantics, preconditions are only checked once before the execution of the relation, whereas invariants are checked each time there is an access to a variable. As a result, the subset of valid pairs of instances for a given transformation is built gradually. Valid pairs of instances of metaclasses are those for which consistency checking evaluates to TRUE for all relations. Relations are invoked recursively, starting from top-level relations. Therefore, modeling a consistency rule by means of a QVT-R transformation can be achieved as follows.

Starting from 1, assign a number to each relation invokation. The result is a recursive call tree whose numbering follows the order of a depth-first traversal. Let n be the total number of relation invokations. Let V_i be the set of valid pairs of instances of metaclass tuples after the execution of the i^{th} invoked relation, for $i \in \{1, \ldots, n\}$. In addition, V_0 is the set of valid pairs of instances before the execution of the transformation, that is $V_0 = \text{COND}_{\langle \mathbb{C}_l \rangle} \times \text{COND}_{\langle \mathbb{C}_r \rangle}$.

Given that each relation filters the space of valid pairs, we have $V_0 \supseteq V_1 \supseteq \cdots \supseteq V_n$. At the end of the execution, V_n represents the set of consistent pairs of instances, i.e. a consistency rule. If there exists $i \in \{1, \ldots, n\}$ such that $V_i = \varnothing$, then it is impossible to fulfill the consistency specification. Such a transformation is useless in practice.

As V_i sets are possibly infinite, computing all consistent pairs of instances is never done in practice. The point of this approach is to show that objects similar to consistency rules can be obtained with QVT-R. The normal use of the language is to check one pair of instances at a time, in a particular direction (i.e. with a source model and a target model).

Finally, a consistency specification can be defined as a set of QVT-R transformations. A set of models is consistent if the execution of all transformations associated with them (in `checkonly` mode) return TRUE in all directions of consistency checking. Starting from domain patterns and conditions, this section demonstrated how objects from the consistency framework can be generated using the QVT-R transformation language.

This makes QVT-R a suitable language for writing declarative consistency specifications. This also concludes the mapping between QVT-R concepts and the consistency framework of Section 3.1.

4. Principles of Decomposition

The previous chapter provided a frame of reference to deal with consistency in a set of metamodels, both theoretically and programmatically. It is within this context that the purpose of this thesis, the *decomposition procedure*, is introduced.

This section focuses on the characteristics of the decomposition procedure. The detailed operation of the procedure is explained in Chapter 5.

First, the principle of decomposition of relations is introduced. Its relevance to multi-model consistency preservation is also justified. Second, three methods to achieve decomposition are presented. Finally, important properties of the decomposition procedure such as conservativeness are discussed, in particular through the definition of invariants about inputs and outputs of the procedure.

4.1. Introduction to the Decomposition Procedure

From now on, a consistency specification refers to a set of consistency rules or a set of QVT-R transformations depending on the desired perspective on consistency. A first (informal) definition of the decomposition of relationships is as follows:

Definition 4.1.1 *In consistency management, the* **decomposition of consistency relations** *is an optimisation technique that transforms a consistency specification into a simpler but equivalent consistency specification. The algorithm resulting from this optimization technique is called the* **decomposition procedure**.

Figure 4.1.: Schematic view of the decomposition procedure

In other words, the decomposition procedure takes a set of metamodels and a consistency specification as input and *decomposes* consistency relations to detect independent, redundant or unnecessary consistency requirements.

The fact that the procedure is an optimisation technique means that it acts on consistency specifications but it does not achieve consistency preservation itself. More precisely, the

decomposition procedure does not perform consistency checking and, consequently, does not execute QVT-R transformations. Consistency specification analysis only requires read-only access to metamodels. For this reason, metamodel instances are also ignored by the decomposition procedure.

It is now necessary to define what "equivalent" consistency specifications are, as well as what a "simpler" consistency specification is.

4.1.1. Equivalent Consistency Specifications

The decomposition procedure must fulfill some important conditions in order to ensure that it does not alter consistency preservation when updating consistency specifications. The first condition is that consistency specifications at the beginning and at the end of the algorithm must be equivalent.

Definition 4.1.2 *Let $\underline{\mathcal{R}}_1$ and $\underline{\mathcal{R}}_2$ be two consistency specifications on a set of metamodels $\mathcal{M}$. Let $\mathcal{I}_\mathcal{M}$ be the set of all possible instances of metamodels of $\mathcal{M}$.*
$\underline{\mathcal{R}}_1$ and $\underline{\mathcal{R}}_2$ are said to be **equivalent***, denoted by $\underline{\mathcal{R}}_1 \equiv \underline{\mathcal{R}}_2$, if:*

$$\forall \mathcal{I} \in \mathcal{I}_\mathcal{M} : \mathcal{I} \text{ is } \underline{\mathcal{R}}_1\text{-consistent} \iff \mathcal{I} \text{ is } \underline{\mathcal{R}}_2\text{-consistent}$$

Equivalence is simply ensured by the set of valid pairs of metamodel instances. There must be no pair of instances that was not consistent at first but becomes consistent after the execution of the procedure. Conversely, all consistent pairs of instances must stay consistent according to the resulting consistency specification.

4.1.2. Complexity of Consistency Specifications

To assess the efficiency of the decomposition procedure, it is necessary to ensure that the resulting consistency specification is in some ways simpler or easier to apply than the initial one. This subsection explains on which criteria it is possible to distinguish two specifications operating on the same set of metamodels, which of these criteria are important in the context of the decomposition procedure and how to use them to gauge the results of the procedure.

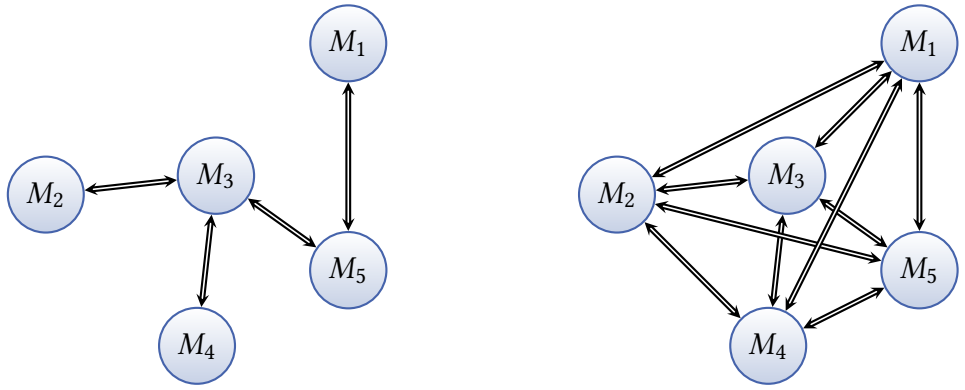
In an introduction to multi-model consistency preservation [Kla18], Klare identified four major challenges regarding the development of binary transformations in a set of metamodels. These challenges are:

- *Compatibility*, i.e. there are no contradictory consistency relations to check or enforce consistency between two metamodels;
- *Modularity*, i.e. omitting some metamodels does not prevent to check or enforce consistency for those remaining;
- *Comprehensibility*, i.e. there is no need to combine many consistency relations to check consistency between two metamodels;

- *Evolvability*, i.e. the definition of new consistency relations requires little effort.

An important point is that it is impossible to fully meet all four challenges with one consistency specification. This is a consequence of the use of combinations of binary relations rather than n-ary relations, as explained in Section 3.1.1. Some implementation choices must be made during the development of consistency relations. These choices are known as *specification trade-offs*.

The most appropriate way to visualize these necessary trade-offs is to observe the structure of various consistency specifications, that is, the *topology* of their consistency relation graphs. The fulfillment of these challenges strongly depends on the density of the consistency relation graph.



(a) Tree topology (sparse) (b) Complete graph topology (dense)

Figure 4.2.: Two topologies with an extremal density, adapted from [Kla18]

Let $\{M_1, \ldots, M_5\}$ be the set of metamodels depicted in Figure 4.2. Following the example of [Kla18], it is possible to choose two consistency specifications such that their density is extremal. These specifications address different challenges.

In Figure 4.2a, the consistency relation graph is a tree. Therefore, the density of the graph is *minimal*. It inherently provides compatibility. As there is only one path between any two metamodels of the graph, there is only one way to preserve consistency and thus, no contradiction. However, such specifications do not evolve easily. In the worst case, adding a metamodel to the tree requires to update all existing relations. Moreover, a tree topology is hardly modular. Removing a metamodel that is not a leaf of the tree breaks consistency for some other metamodels. Nor does the specification foster comprehensibility, especially if one metamodel is the root of the tree whereas the other is a leaf.

In Figure 4.2b, the consistency relation graph is a complete graph. Therefore, the density of the graph is *maximal*. Because of the number of paths between any two metamodels, it is hard to ensure compatibility. Also, adding a new metamodel to the specification may require the development of consistency relations with all existing metamodels, making it difficult to evolve specifications. On the contrary, modularity is optimal: any induced subgraph of a complete graph is still a complete graph. Finally, comprehensibility is high

Challenge	Influence of the decomposition procedure
Compatibility	Detection of possible incompatibilities
Modularity	No change
Comprehensibility	Minor decrease
Evolvability	Minor improvement

Table 4.1.: Summary of the influence of the decomposition procedure

because all the information sought for consistency between two metamodels belongs to only one consistency relation.

Of all these challenges, compatibility is probably the most important one. A reason for this is that compatibility is a functional property, i.e. the applicability of the specification directly depends on whether relations are compatible or not, whereas other properties are non-functional. Incompatible consistency relations prevent consistency preservation. If two consistency relations differ on the state in which a consistent model should be, they can be executed in turn without ever reaching a final consistent state.

As a result, sparse (also called *tree-like*) consistency relation graphs are preferred. Note that extreme topologies presented in Figure 4.2 are rarely encountered in practice. In most cases, consistency specifications are in-betweens that tries to maximize both compatibility and modularity. Defining tree-like topologies of consistency relation graphs is complicated. This requires finding a small number of metamodels that contain enough information to define consistency relations whose combination can express consistency between any two metamodels.

More reasonably, another approach for consistency specification is as follows. First, developers can define consistency relations without addressing redundancy issues. Consequently, the resulting consistency relation graph tends to be dense. Then, the specification can be optimized in such a way that redundancies are removed (which may lead to the removal of some consistency relations) and independent subsets of consistency relations are detected. This optimization transforms the topology of the initial specification into a set of tree-like topologies.

The decomposition of consistency relations is a procedure whose purpose is to achieve this optimization. Regarding challenges identified above, decomposition fosters *compatibility* (inherent in the resulting tree-like structures) as well as *evolvability*. The reason for this is that it detects insofar as possible redundancy and therefore possible incompatibilities. The fewer incompatibilities, the more compatibility is achieved. Moreover, it has little impact on *modularity*: removed relations (due to redundancy) can always be recreated and subsets of independent relations indicate on which other metamodels the consistency of a given metamodel depends.

4.2. Means of Decomposition of Specifications

This section investigates changes that can be made to consistency specifications to meet the purpose of the decomposition procedure. In concrete terms, the goal is to find a way

to transform a specification into another specification whose consistency relation graph is made up of subgraphs of independent consistency relations.

First, three avenues for decomposition are presented. The first and most obvious consists in isolating connected components of the consistency relation graph. The second consists in removing (redundant) consistency relations that can be replaced by combinations of other relations. The third idea generalizes the second. Assuming that consistency relations can be associated with consistency rules that generate them (see Section 3.1.3), it consists in removing parts of consistency relations, i.e. consistency rules, that can be replaced by combinations of other relations to make remaining rules independent of others.

Then, the last subsection explains how these three ideas can be combined to produce a new consistency specification.

4.2.1. Independent Consistency Subgraphs

The first way to find subsets of independent consistency relations in a consistency relation graph is to look at connected components of the graph. The reason for this is that two subsets that do not share a consistency relation cannot be interrelated.

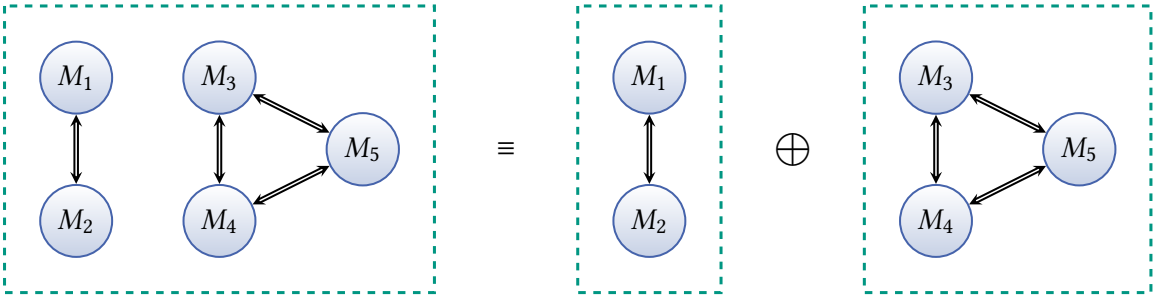


Figure 4.3.: Independent consistency subgraphs

In Figure 4.3, the $\oplus$ symbol denotes the disjoint union of two graphs. More generally, any graph can be reconstructed from the disjoint union of its connected components.

This optimization is the easiest and the first that should be performed on a consistency relation graph. The following two methods can then be performed in each connected component of the initial graph. Then, the goal is that each connected component tends to become a tree-like structure.

4.2.2. Totally Redundant Consistency Relations

This subsection focuses on *redundant* consistency relations; that is, rules whose existence in the specification does not change the set of consistent instances.

This may occur when the application of rules of the redundant relation and the application of rules of a combination of other consistency relations result in the same set of consistent instances. In this case, the redundant relation can be removed from the graph, thus proving that the relation was compatible with others.

The following definitions formalize this principle. As an exhaustive description of consistency is now required, definitions use consistency rules. We denote $\underline{\mathcal{R}}_C$ the set of consistency rules that generate the consistency relation C. Moreover, a pair of models *fulfills* a consistency relation C if it fulfills all consistency rules in $\underline{\mathcal{R}}_C$.

Definition 4.2.1 *Let $\underline{\mathcal{R}}$ be a consistency specification. A* **combination of consistency relations** *is a path in the consistency relation graph of $\underline{\mathcal{R}}$.*

In other words, a combination of consistency relations is a sequence of distinct metamodels that are all joined by a consistency relation. It is now possible to extend the notion of consistency rule fulfillment to combinations.

Definition 4.2.2 *Let $\mathcal{M}$ be a set of metamodels and let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$ whose consistency relation graph is connected. Let $\underline{C} = (M_1, \ldots, M_i, M_{i+1}, \ldots, M_n)$ with $1 < i < n$ be a combination of relations between two metamodels M_1 and M_n.*

The tuple of instances $(I_{M_1}, \ldots, I_{M_n}) \in \mathcal{I}_{M_1} \times \cdots \times \mathcal{I}_{M_n}$ **fulfills the combination** *$\underline{C}$ of consistency relations if and only if:*

$$\bigwedge_{i=1}^{n-1} \left(\{I_{M_i}, I_{M_{i+1}}\} \textit{ fulfills } \underline{\mathcal{R}}_{C_{i,i+1}} \right)$$

The set of consistent instances according to a consistency relation can now be compared with combinations of consistency relations to find out if it is redundant.

Definition 4.2.3 *Let $\mathcal{M}$ be a set of metamodels. Let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$ whose consistency relation graph is connected.*

A consistency relation $C_{l,r}$ between two metamodels M_l and M_r is **totally redundant** *if there exists a set $\mathfrak{C}$ of combinations from M_l to M_r such that:*

$$\{\{I_{M_1}, I_{M_n}\} \mid \forall\, I = (I_{M_1}, \ldots, I_{M_n}) \in \mathcal{I}_{M_1} \times \cdots \times \mathcal{I}_{M_n},\ \exists\, \underline{C} \in \mathfrak{C} : \left(I \textit{ fulfills } \underline{C}\right)\} \tag{4.1}$$

$$= \{\{I_{M_l}, I_{M_r}\} \mid \forall I_{M_l} \in \mathcal{I}_{M_l}, \forall I_{M_r} \in \mathcal{I}_{M_r} : \{I_{M_l}, I_{M_r}\} \textit{ fulfills } \underline{\mathcal{R}}_{C_{l,r}}\} \tag{4.2}$$

The equality above holds if both sets contain the same consistent pairs of instances. In Term (4.1), the set contains all consistent pairs of models of M_l and M_r if these models were part of an instance that fulfills at least one combination of relations. That is, these are the consistent models at the endpoints of at least one combination once the consistency was preserved for all metamodels of all combinations.

In Term (4.2), the set contains all pairs of instances that fulfill the possibly redundant consistency relation $C_{l,r}$. If both equations return the same set, $C_{l,r}$ performs the same task as the set $\mathfrak{C}$ of combinations of consistency relations and can therefore be removed.

In Figure 4.4, the relation $C_{1,4}$ can be removed if it can be replaced by the combination (M_1, M_2, M_3, M_4). In this case, $C_{1,4}$ is said to be totally redundant. This means that

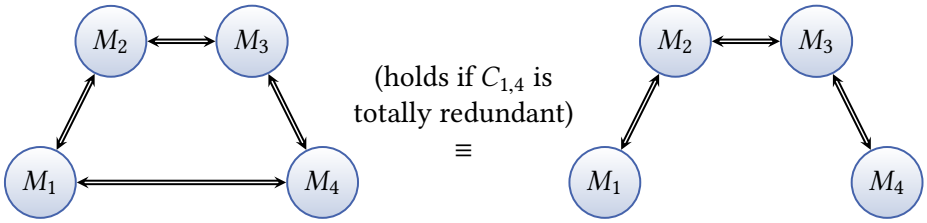


Figure 4.4.: Removal of a totally redundant consistency relation

all consistent pairs of instances of M_1 and M_4 according to $C_{1,4}$ are the same as those obtained by applying relations $C_{1,2}$, $C_{2,3}$ and $C_{3,4}$ and only retaining tuples of models of (M_1, M_2, M_3, M_4) for which these relations are fulfilled.

Ultimately, totally redundant consistency relations are relations in which *all* consistency rules can be replaced by alternative combinations of consistency relations. In that case, these rules are unnecessary and ce be removed from the specification. Their removal also leads to the removal of the consistency relation they generated.

4.2.3. Partially Redundant Consistency Relations

Removing totally redundant consistency relations is a significant optimisation. However, this rarely happens in practice. The third avenue for decomposition presents a generalized case for the detection of independent consistency relations.

The basic idea is that compatibility can still be improved if combinations of consistency relations can replace some rules of a consistency relation with the same endpoints, not necessarily all of them. In this case, the consistency relation is *decomposed*. To check consistency rules of a relation separately, we define what a combination of consistency rules is.

Definition 4.2.4 *Let $\underline{\mathcal{R}}$ be a consistency specification. A* **combination of consistency rules** *is a sequence $\underline{CR}$ of consistency rules $(\mathcal{R}_{\mathbb{C}_1,\mathbb{C}_2}, \ldots, \mathcal{R}_{\mathbb{C}_{i-1},\mathbb{C}_i}, \mathcal{R}_{\mathbb{C}_i,\mathbb{C}_{i+1}}, \ldots, \mathcal{R}_{\mathbb{C}_{n-1},\mathbb{C}_n})$ such that two incident rules have a common metaclass tuple in their definition.*

In other words, a combination of rules represents the selection of one consistency rule in each relation of a combination of consistency relations, under the condition that two rules relating the same metamodel are defined with the same metaclass tuple.

Definition 4.2.5 *Let $\underline{CR} = (\mathcal{R}_{\mathbb{C}_1,\mathbb{C}_2}, \ldots, \mathcal{R}_{\mathbb{C}_{i-1},\mathbb{C}_i}, \mathcal{R}_{\mathbb{C}_i,\mathbb{C}_{i+1}}, \ldots, \mathcal{R}_{\mathbb{C}_{n-1},\mathbb{C}_n})$ be a combination of relations between two metamodels M_l and M_r. The combination $\underline{CR}$* **replaces** *consistency rule $\mathcal{R}_{\mathbb{C}_l,\mathbb{C}_r}$ if and only if $\mathbb{C}_1 = \mathbb{C}_l$ and $\mathbb{C}_n = \mathbb{C}_r$ and:*

$$\{\{I_{M_1}, I_{M_n}\} \mid \forall\, I = (I_{M_1}, \ldots, I_{M_n}) \in \mathcal{I}_{M_1} \times \cdots \times \mathcal{I}_{M_n} : \big(I \textit{ fulfills } \underline{CR}\big)\} \tag{4.3}$$

$$= \{\{I_{M_l}, I_{M_r}\} \mid \forall I_{M_l} \in \mathcal{I}_{M_l}, \forall I_{M_r} \in \mathcal{I}_{M_r} : \{I_{M_l}, I_{M_r}\} \textit{ fulfills } \mathcal{R}_{\mathbb{C}_l,\mathbb{C}_r}\} \tag{4.4}$$

Let $\mathcal{R}$ be a rule of this consistency relation. There are three possible scenarios:

1. $\mathcal{R}$ can be replaced by an alternative combination of rules. Therefore, $\mathcal{R}$ can be *removed* from the consistency specification.

2. $\mathcal{R}$ cannot be replaced because there exist alternative combinations of rules but none of them has the same set of consistent instances as $\mathcal{R}$. Therefore, $\mathcal{R}$ may be *contradictory* to other rules.

3. $\mathcal{R}$ cannot be replaced because there does not exist an alternative combination of rules. As a result, $\mathcal{R}$ is *independent* of other rules.

Consequently, reasoning about consistency rules rather than consistency relations offers more possibilities to detect possible incompatibilities in a consistency specification. Inside a single relation, rules may be either redundant, independent or in conflict with other combinations of rules. The decomposition of the relation consists in identifying the nature of each rule in the relation. A partially redundant consistency relation is a relation in which at least one rule can be replaced.

Definition 4.2.6 *Let $\mathcal{M}$ be a set of metamodels. Let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$ whose consistency relation graph is connected. Let $\mathfrak{CR}$ be a set of combinations of rules between two metamodels M_l and M_r.*

A consistency relation C between M_l and M_r generated by a set of consistency rules $\underline{\mathcal{R}}_C$ is **partially redundant** *if and only if:*

$$\exists\, \mathcal{R} \in \underline{\mathcal{R}}_C, \exists\, \underline{CR} \in \mathfrak{CR} : \underline{CR} \text{ replaces } \mathcal{R}$$

Definition 4.2.3 can be recreated by replacing the first existential quantifier in Definition 4.2.6 by a universal quantifier, i.e. $\forall\, \mathcal{R} \in \underline{\mathcal{R}}_C$. As a result, Definition 4.2.6 is a relaxed version of Definition 4.2.3. Independent consistency rules also foster compatibility in a specification since they generate separate consistency relations which represent trivial tree graphs, i.e. only one edge.

In Figure 4.5, suppose that the relation $C_{2,3}$ is partially redundant. Such a decomposition can be obtained if $C_{2,3}$ was generated by at least one redundant consistency rule and exactly one independent consistency rule. If a consistency rule is redundant, i.e. it can be replaced

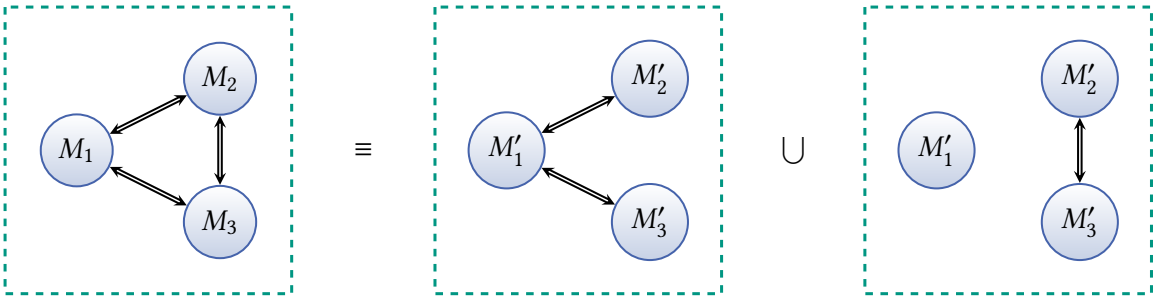


Figure 4.5.: Decomposition of a partially redundant consistency relation

	Decomposition method	Description
(I)	Independent subgraphs	Computation of connected components
(II)	Totally redundant relations	Removal of consistency relations
(III)	Partially redundant relations	Decomposition of consistency relations

Table 4.2.: Summary of decomposition methods

by a combination of rules derived from $(C_{2,1}, C_{1,3})$. If a consistency rule is independent, it can be separated from others and form a new independent subgraph. Consequently, the first resulting subgraph On the contrary, the second consistency rule is independent, i.e. there is no

Contrary to the result of the method of independent consistency subgraphs (Section 4.2.1) that forms a partition ($\oplus$) of the initial graph, the decomposition of partially redundant relations results in a *non-disjoint* union ($\cup$) of subgraphs. The reason is that the (newly independent) decomposed relation and the combination share the same endpoints.

4.2.4. Towards a Decomposition Procedure

Previous subsections introduced methods to show that there exist several means to detect redundancy and incompatibilities in a consistency specification. These methods can be combined to lay the foundations for a decomposition procedure.

4.2.4.1. Combining Decomposition Methods

The first method applies on the consistency relation graph and does not require knowing consistency rules. The last two methods apply on connected consistency relation graphs and require knowing consistency rules.

Moreover, the application of the last two methods cannot produce disconnected graphs. This is because if a relation is redundant, there must exist a combination of other consistency relations with the same endpoints. If the relation is totally redundant, then it is removed and the graph is not disconnected because the combination still exists. If the relation is partially redundant, then the relation is not removed and the graph is not disconnected either. Consequently, the methods can be chained as follows.

1. Apply the first method to transform the consistency relation graph into a set of connected components;

2. For each connected component, apply the second and the third methods (both work the same way) on edges to detect if they are redundant.

The resulting consistency specification is a set of independent connected components, which are themselves decomposed as sets of independent consistency relations. Connected components take the form of *tree-like* structures. Figure 4.4 and Figure 4.5 give an overview of what a tree-like structure is. When a relation is removed from a component, the

component remains connected but tends towards a tree. In the case of partial redundancy, the newly independent relation also forms a tree with two vertices.

4.2.4.2. Algorithms for Decomposition Methods

In view of the decomposition methods, the decomposition procedure can be considered as an implementation of these methods on QVT-R specifications.

The first method can be implemented by means of a computation of connected components in the consistency relation graph. The two other methods are based on a unique algorithm. Whether the redundancy is then total (second method) or partial (third method) depends on the result of the algorithm.

The second and the third methods operate by comparing two sets containing consistent instances. However, these sets are possibly infinite, meaning that an exhaustive comparison is impossible in general. Moreover, the decomposition procedure works with metamodels and QVT-R specifications only. Consequently and as stated in Section 3.2, consistency rules use intensional definitions, i.e. definitions based on elements of metamodels rather than explicit instances.

The consequence for the decomposition procedure is that reasoning on sets of instances must be achieved by reasoning on domain patterns and OCL expressions. In other words, an algorithm for the second and the third methods must statically analyze QVT-R relations and therefore use formal methods.

4.3. Formal Properties

Requirements expressed in previous sections can be translated into properties of the decomposition procedure. Therefore, it is possible to give a formal meaning to the decomposition procedure and to impose conditions on results of the procedure.

There are two important properties. First, *conservativeness* to ensure that the resulting specification preserves consistency in the same way as the original specification. Second, *usefulness* to ensure that the procedure finds the decomposition of the original specification if it exists. In the following subsections, let DECOMP denote the decomposition procedure, i.e. a function that takes a set of metamodels and a consistency specification on this set, and returns another consistency specification.

4.3.1. Conservativeness

Given that the decomposition procedure is an optimization technique for consistency specifications, it is important to check that optimizations do not alter the specification. Therefore, the set of consistent models must not be altered by the decomposition procedure. This property is called *conservativeness*.

Property 4.3.1 *(Conservativeness) Let $\mathcal{M}$ be a set of metamodels, let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$. The decomposition procedure is* **conservative** *if and only if:*

$$\underline{\mathcal{R}'} = \text{DECOMP}(\underline{\mathcal{R}}, \mathcal{M}) \implies \underline{\mathcal{R}} \equiv \underline{\mathcal{R}'}$$

The consequence of conservativeness is that optimizations should only be performed with the guarantee that the specification will not be altered. A proof of conservativeness of the decomposition procedure will be detailed in Chapter 6 and discussed in Chapter 7.

4.3.2. Usefulness

The resulting specification being equivalent to the initial one is not enough to ensure that the procedure improved the applicability of the consistency specification. To this purpose, we need to introduce another property, called *usefulness*.

The improvement made by the decomposition procedure on a consistency specification depends on the structure of the resulting consistency relation graph. In the best case, the consistency relation graph is a disjoint union of trees.

In the following definitions, $V_{\mathcal{G}}$ (resp. $E_{\mathcal{G}}$) denotes the set of vertices (resp. edges) of a graph $\mathcal{G}$. First, the notion of decomposition is formalized. Then, a value is associated with each decomposition to allow comparisons. Finally, usefulness is defined in order to assert that the decomposition procedure improves consistency specifications.

Definition 4.3.1 *Let $\underline{\mathcal{R}}$ be a consistency specification and $\mathcal{G}$ its graph. A* **decomposition** *of $\underline{\mathcal{R}}$, denoted $\mathcal{D}_{\underline{\mathcal{R}}}$, is a finite set $\{\mathcal{G}_i\}_{1 \leq i \leq n}$ of subgraphs of $\mathcal{G}$ such that:*

(1) $\forall i \neq j : E_{\mathcal{G}_i} \cap E_{\mathcal{G}_j} = \varnothing$

(2) $\sum_{i=1}^{n} |E_{\mathcal{G}_i}| \leq |E_{\mathcal{G}}|$

(3) $V_{\mathcal{G}} = \bigcup_{i=1}^{n} V_{\mathcal{G}_i}$

In other words, a decomposition provides information on the output of the decomposition procedure. It is a set of independent subgraphs. To this end, three conditions must be fulfilled. First, an edge cannot appear in several graphs. The reason is that two relations of distinct subgraphs are independent. Then, there cannot be more edges in the decomposition than in the original graph because the procedure does not create edges. Finally, all metamodels of the original consistency relation graph must appear in the decomposition given that the decomposition procedure does not affect metamodels.

Note that a consistency specification may correspond to several decompositions. The expected decomposition is the one returned by the procedure, as shown on Figure 4.1 (p. 35). It is built during the execution of the procedure by isolating connected components and independent consistency relations as new subgraphs ($\mathcal{G}_i$).

To provide a point of comparison with non-decomposed specifications (e.g. specifications that serve as inputs of the procedure), we can also define a trivial decomposition. The *trivial decomposition* for a specification $\underline{\mathcal{R}}$ is a singleton $\{\mathcal{G}\}$ where $\mathcal{G}$ is the consistency relation graph of $\underline{\mathcal{R}}$. By convention, the decomposition of a specification that is not the result of the decomposition procedure is the trivial decomposition.

Definition 4.3.2 *Let $\mathcal{D}_{\underline{\mathcal{R}}} = \{\mathcal{G}_i\}_{1 \leq i \leq n}$ be a decomposition of $\underline{\mathcal{R}}$. The* **decomposition size** *of $\mathcal{D}_{\underline{\mathcal{R}}}$, denoted $\overline{\mathcal{D}_{\underline{\mathcal{R}}}}$, is defined as the mean of the number of edges in all subgraphs, i.e.*

$$\overline{\mathcal{D}_{\underline{\mathcal{R}}}} = \overline{\{\mathcal{G}_i\}_{1 \leq i \leq n}} = \frac{1}{n} \sum_{i=1}^{n} |E_{\mathcal{G}_i}|$$

Intuitively, the more a specification is scattered in small subsets of independent consistency relations, the smaller its decomposition size is. The decomposition size of a consistency specification that is not the result of the decomposition procedure is equal to the number of edges of its consistency relation graph.

Property 4.3.2 *(Usefulness) Let $\mathcal{M}$ be a set of metamodels, let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$. The decomposition procedure is* **useful** *if and only if:*

$$\underline{\mathcal{R}'} = \textsc{decomp}(\underline{\mathcal{R}}, \mathcal{M}) \implies \overline{\mathcal{D}_{\underline{\mathcal{R}'}}} \leq \overline{\mathcal{D}_{\underline{\mathcal{R}}}}$$

This property implies that whatever the input specification, the output specification of the decomposition procedure will be either identical ($\overline{\mathcal{D}_{\underline{\mathcal{R}'}}} = \overline{\mathcal{D}_{\underline{\mathcal{R}}}}$) or more applicable ($\overline{\mathcal{D}_{\underline{\mathcal{R}'}}} < \overline{\mathcal{D}_{\underline{\mathcal{R}}}}$). In the latter case, the result of the decomposition is called *positive*.

The conjunction of conservativeness and usefulness leads to an equivalent, yet more applicable consistency specification. It ensures at once that the resulting specification is equivalent to the initial one and that there exists a decomposition which make independent consistency relations explicit.

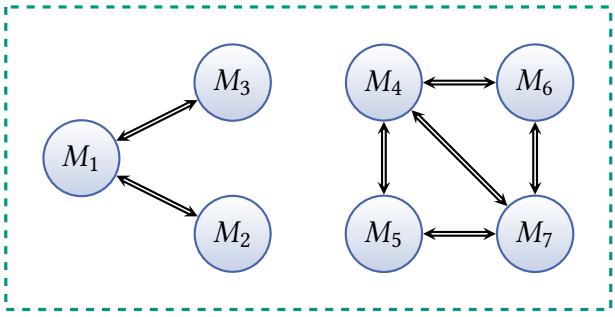


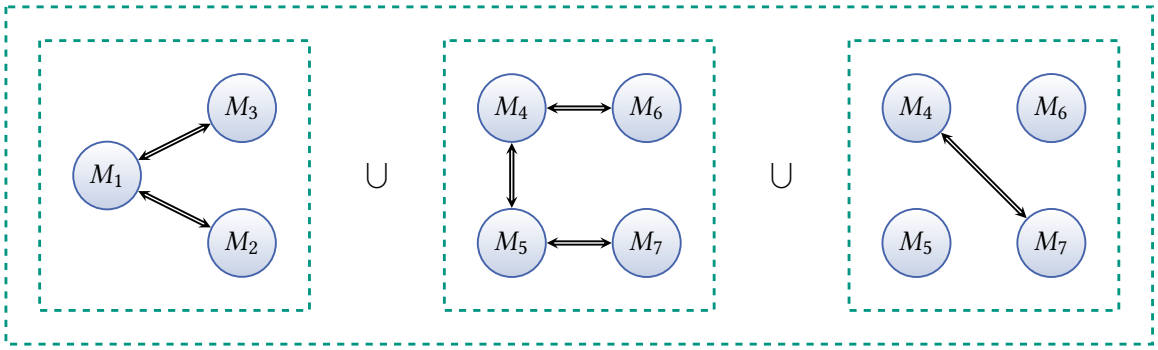
Figure 4.6.: Consistency specification for Example 4.3.1

Example 4.3.1 *Let $\mathcal{M} = \{M_1, \ldots, M_7\}$ be a set of metamodels. Let $\underline{\mathcal{R}}$ be a consistency specification on $\mathcal{M}$ whose consistency relation graph is depicted in Figure 4.6.*

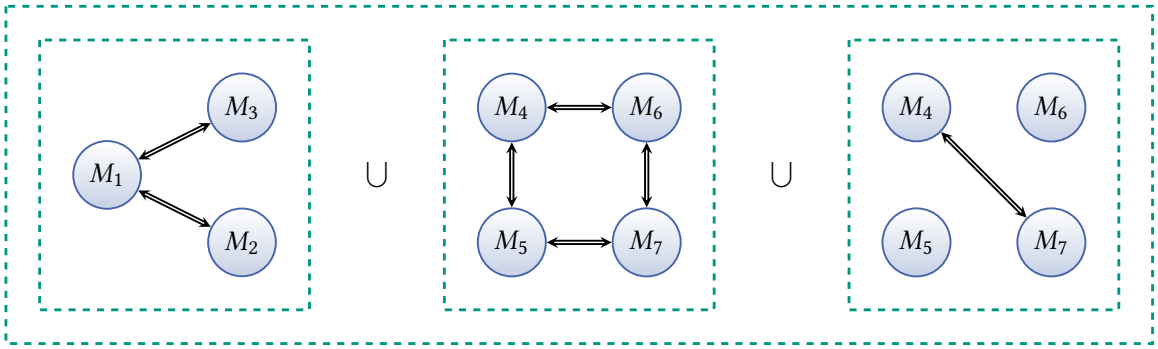
Figure 4.7 represents three possible decompositions of $\underline{\mathcal{R}}$ with their respective decomposition sizes. Decomposition $\mathcal{D}_1$ is optimal because every subgraph is a forest (i.e. a disjoint set of trees). Its size is slightly less than the size of $\mathcal{D}_2$ in which the consistency relation between M_6 and M_7 was not removed. Even if it is not optimal, $\mathcal{D}_2$ remains a valid decomposition.

Decomposition $\mathcal{D}_3$ consists of one subgraph which is the consistency relation graph of $\underline{\mathcal{R}}$. In other words, $\mathcal{D}_3$ is the trivial decomposition of $\underline{\mathcal{R}}$, i.e. the one that applies if the specification is not a result of the decomposition procedure. If the procedure returns $\mathcal{D}_1$ or $\mathcal{D}_2$, the result is positive because $2 < 7$ and $2/3 < 7$.

Decomposition $\mathcal{D}_1$
— $\mathcal{D}_1$ is the optimal decomposition of $\underline{\mathcal{R}}$
— Decomposition size: 2



Decomposition $\mathcal{D}_2$
— Decomposition size: 7/3



Decomposition $\mathcal{D}_3$
— $\mathcal{D}_3$ is the trivial decomposition of $\underline{\mathcal{R}}$
— Decomposition size: 7

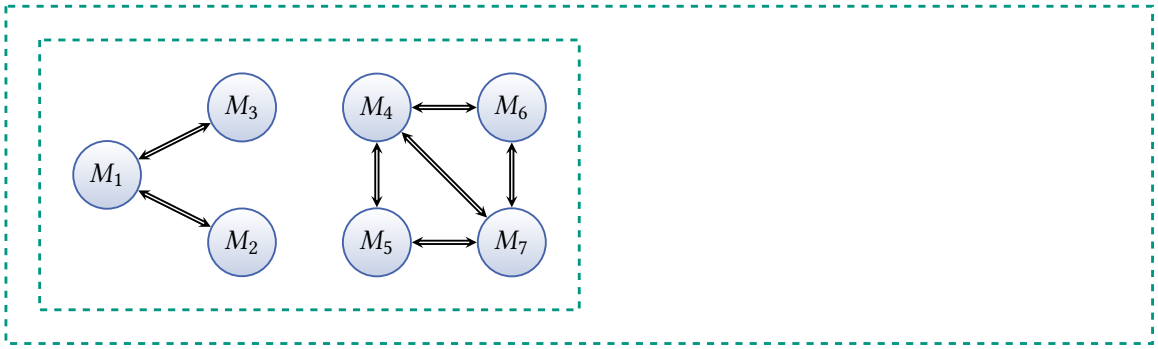


Figure 4.7.: Some possible decompositions of Example 4.3.1 with their sizes

5. Decomposition Procedure

The decomposition procedure offers an implementation of decomposition methods discussed in Section 4.2. The implementation takes a consistency specification (represented as a set of QVT-R transformations) and a set of metamodels as inputs and returns a valid decomposition of the specification.

First, this chapter presents some important considerations for a function implementation. In particular, it introduces an intermediate and appropriate representation of consistency specifications, the *metagraph*. This leads to an outline of the decomposition procedure, then divided into two parts. The first part explains how a consistency specification can be encoded in a metagraph. The second part describes the use of metagraphs to generate a decomposition of consistency relations. Finally, the last section discusses the validation of formal properties identified in Section 4.3 by the implementation.

5.1. Tractable Consistency Relations

Decomposition methods provide a means to decide if consistency relations are redundant or independent. As mentioned in Section 4.2.4, these methods have both advantages and shortcomings. On one hand, composition of decomposition methods leads to a generic decomposition procedure for consistency specifications. On the other hand, the complexity of these methods lies in the algorithms required to implement them.

This section presents a representation of tractable consistency relations, i.e. a way to combine the benefits of consistency relations and consistency rules in a unique data structure for an efficient representation of consistency specifications. The aim is to generate consistency relations and combinations of relations from QVT-R transformations.

5.1.1. Two Aspects of Consistency Specifications

As of now, consistency specifications have been presented in two different ways. First, by way of *consistency relations*, i.e. edges of consistency relation graphs. This aspect of consistency can be described as *global*. Consistency relations are especially useful in the context of multi-model consistency preservation because they emphasize interrelated metamodels. Additionally, the first decomposition method – the retrieval of independent connected subgraphs – relies on the topology of the consistency specification.

However, having a set of metamodels bound by consistency relations is not enough to ensure consistency. In that case, another aspect of consistency is essential: a *local* aspect described by *consistency rules*. Given that consistency is a property of models, consistency rules define which pairs of models are consistent, whereas consistency relations only indicate that metamodels contain redundant information.

These two approaches complement one another. As stated in Section 3.1.3, a consistency relation is associated with one or more consistency rules.

This is illustrated in Figure 5.1. The upper part of the figure represents a relation between two metamodels P and E in a consistency relation graph. Another viewpoint, defined by a consistency rule, shows that this relation consists of two metaclasses with a constraint on their *name* attributes.

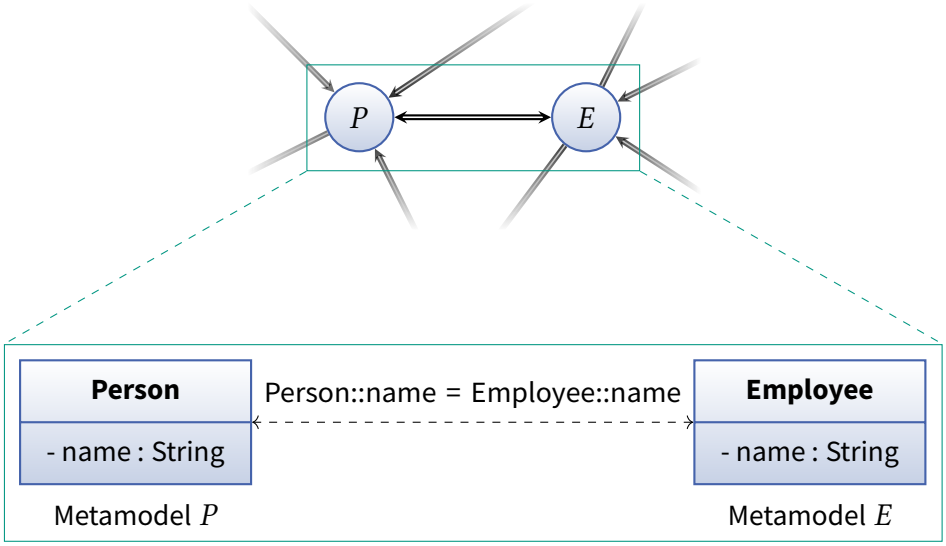


Figure 5.1.: From consistency relation graph to consistency rule

An important consideration for an implementation of the decomposition procedure is to find an appropriate representation for consistency specifications. This representation must be able to express both aspects of specifications at once. In Section 3.4, we showed how the formalism of consistency rules can be reproduced using QVT-R transformations. Therefore, QVT-R covers at least the local aspect of consistency specifications.

To cover the global aspect of specifications with an appropriate data structure, we need a way to relate QVT-R transformations with combinations of consistency relations and graph topologies. A first approach is to define a new kind of graph using metamodel elements as vertices and consistency rules as edges. In doing so, consistency is now entirely defined at the metamodel element level rather than being split between metamodels and metamodel elements. Therefore, specifications for the decomposition procedure are represented with a finer granularity.

This paradigm change requires a redefinition of concepts introduced in previous chapters and related to consistency relation graphs such as combinations of relations or tree-like structures. Regarding graphs at the metamodel level, Section 3.1.1 highlighted the fact that graphs (i.e. binary relations) were a sufficiently expressive formalism compared with hypergraphs (i.e. n-ary relations). This is no longer the case when consistency is expressed at the metamodel element level. Example 5.1.1 is an example of a consistency rule that cannot be expressed with binary relations. This motivates the need for hypergraphs in the decomposition procedure.

```
1     top relation ResidentEmployee {
2         fstn: String;
3         lstn: String;
4
5         domain pers r:Resident {firstname=fstn, lastname=lstn};
6         domain emp e:Employee {name=fstn + '␣' + lstn};
7     }
```

Listing 5.1: A ternary consistency rule with QVT-R

Example 5.1.1 *Consider the QVT-R relation* ResidentEmployee *in Listing 5.1 which binds two metaclasses,* Resident *and* Employee*. The* Resident *metaclass has two attributes:* firstname *and* lastname*. The* Employee *metaclass has one attribute:* name*.*

The consistency rule indicates that for each Resident *instance, there should exist an* Employee *instance whose name is the concatenation of the firstname and the lastname of the* Resident *and vice versa. The consistency rule necessarily involves three attributes.*

It can be written as a ternary relation ResidentEmployee(firstname, lastname, name) *such that* ResidentEmployee $\subseteq$ *String* $\times$ *String* $\times$ *String. Modeled as an edge, this relation is actually a ternary edge, i.e. an hyperedge, whose endpoints are attributes of the two metaclasses.*

Definitions of conditions and consistency rules do not impose any restrictions regarding properties of metaclasses. Consistent pairs of objects can therefore be chosen arbitrarily. In the case of QVT-R relations, they are chosen by defining as many property template items as necessary. Consequently, edges can link an arbitrary number of metamodel elements, hence the need for hypergraphs.

Following this approach, vertices of hypergraphs are metamodel elements. As a result, an hyperedge linking several elements only indicates that these elements participate in a consistency rule – just as a consistency relation only indicates that two metamodels are interrelated. For this reason, the consistency rule (or, comparably in an implementation, the QVT-R constraint) is associated with its hyperedge.

5.1.2. Metagraph

The data structure that meets previous requirements for an implementation of the decomposition procedure is called a *metagraph*. It is made up of a hypergraph and a labeling, each hyperedge being labeled with the consistency rules that it must fulfill.

Definition 5.1.1 *Let* $\mathcal{M}$ *be a set of metamodels and let* $\underline{\mathcal{R}}$ *be a consistency specification on* $\mathcal{M}$*. A* **metagraph** *for* $\underline{\mathcal{R}}$ *is a couple* $(\mathcal{H}, c)$ *such that* $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H})$ *is a hypergraph and* $c : E_\mathcal{H} \to \mathcal{P}(\underline{\mathcal{R}}) \setminus \{\varnothing\}$ *is a hyperedge labeling under the following conditions:*

- $V_\mathcal{H}$ *is a set of elements of metamodels of* $\mathcal{M}$ *called* **meta-vertices** *and* $E_\mathcal{H} \subseteq \mathcal{P}(\mathcal{V}_\mathcal{H}) \setminus \{\varnothing\}$ *is a set of hyperedges called* **meta-edges***, i.e. subsets of* $V_\mathcal{H}$*.*

- *c is a function that labels each hyperedge with a nonempty set of consistency rules. When consistency rules are expressed as QVT-R relations, all metamodel elements participating in the definition of the rule must be part of the hyperedge.*

A meta-edge can be labeled by more than one consistency rule. For example, a QVT-R transformation can include two relations whose domain patterns bind the same metamodel elements. In that case, consistency of the meta-edge is decided by the conjunction of all consistency rules.

In the context of QVT-R transformations, the nature of metamodel elements is in fact limited. More precisely, template expressions of domain patterns match either objects (regarding *object template expressions*) or collections of objects (regarding *collection template expressions*). The type of an object is always a metaclass, i.e. an `EClass` in Ecore.

As explained in Section 3.4.2, the selection of consistent objects in QVT-R is achieved by filtering the values that properties of these objects can take. In other words, consistency is the result of conditions applied on object properties. Consequently, metamodel elements are primarily object properties. These properties refer to both attributes (`EAttribute`) and references (`EReference`) of metaclasses (cf. Section 2.2.1 for Ecore).

When a metamodel is instantiated, there are two ways to retrieve the properties of an object: either through its *attributes* or through *role names* of its references. Role names are also known as *association end names*. These two concepts are standard in OCL. Moreover, they appear in the left-hand side of QVT-R *property template items*, in template expressions. Therefore, vertices of metagraphs are either attributes or role names.

An in-depth example of a metagraph and its construction is presented in Section 5.3.

5.1.3. Metagraphs and Constraint Networks

The formalism of metagraphs is actually closely related to *constraint networks*, a major concept in the field of constraint satisfaction (see Section 2.5.1). Although constraint networks are usually applied on well-defined algorithmic problems (such as scheduling problems), they offer a point of comparison with metagraphs.

Accordingly, interesting notions used in the study of constraint networks such as constraint languages and hypertrees will also occur in the study of metagraphs. Table 5.1 provides a comparison between concepts of constraint networks and concepts of multi-model consistency preservation.

Constraint satisfaction	**Consistency specification**
Constraint network	Metagraph
Variable	Metamodel element
Domain	Metamodel element type
Constraint	Meta-edge

Table 5.1.: Similar concepts in constraint satisfaction and consistency preservation

The main difference between constraint satisfaction and consistency preservation lies in the use of structures. Being defined for specific problems, constraint networks rarely need to be updated, unlike consistency specifications. Common methods to solve constraint satisfaction problems such as constraint propagation are not considered here.

In the end, the metagraph provides a convenient way to represent consistency topologies using consistency rules. This representation serves as an input for the implementations of decomposition methods. Moreover, its structure can be related to that of a constraint network. This makes it possible to apply several optimization techniques of constraint satisfaction to facilitate the use of metagraphs.

5.2. Outline of the Decomposition Procedure

In accordance with Section 5.1, the implementation of the decomposition procedure involves the definition and the use of a metagraph, i.e. a single data structure to combine consistency relations and consistency rules. Based on this concept, an outline for the decomposition procedure can be defined as follows.

The procedure is divided into two parts. In the first part, metamodels and QVT-R transformations using these metamodels are parsed from the input. Potential inconsistencies in the input (syntax errors, missing metamodel elements, etc.) are reported to the user. A metagraph is then built by recursively analyzing transformations. The analysis being static, all the information is retrieved from the abstract syntax trees of QVT-R files. The conversion is dynamic, i.e. only metamodel elements that are relevant to consistency are considered, and simultaneous, i.e. global and local aspects of consistency specifications are addressed at the same time.

The construction of the metagraph is depicted in Figure 5.2.

The second part begins when the construction of the metagraph is completed. The aim is then to find independent subsets of consistency relations using the metagraph. To that end, algorithms that implement decomposition methods insofar as possible are introduced. First, the dual of the metagraph is introduced. It is a useful concept to use graph-theoretic algorithms on hypergraphs. Then, an approach to check redundancy by comparing consistency relations and combinations of relations is presented. In particular, the core of the approach is an algorithm that uses a theorem prover to indicate whether a consistency relation can be replaced by a combination of other relations.

The use of the metagraph is depicted in Figure 5.3.

5.3. From Consistency Specification to Metagraph

This section describes the first part of two parts of the decomposition procedure, i.e. an algorithmic construction of a metagraph from a set of QVT-R files.

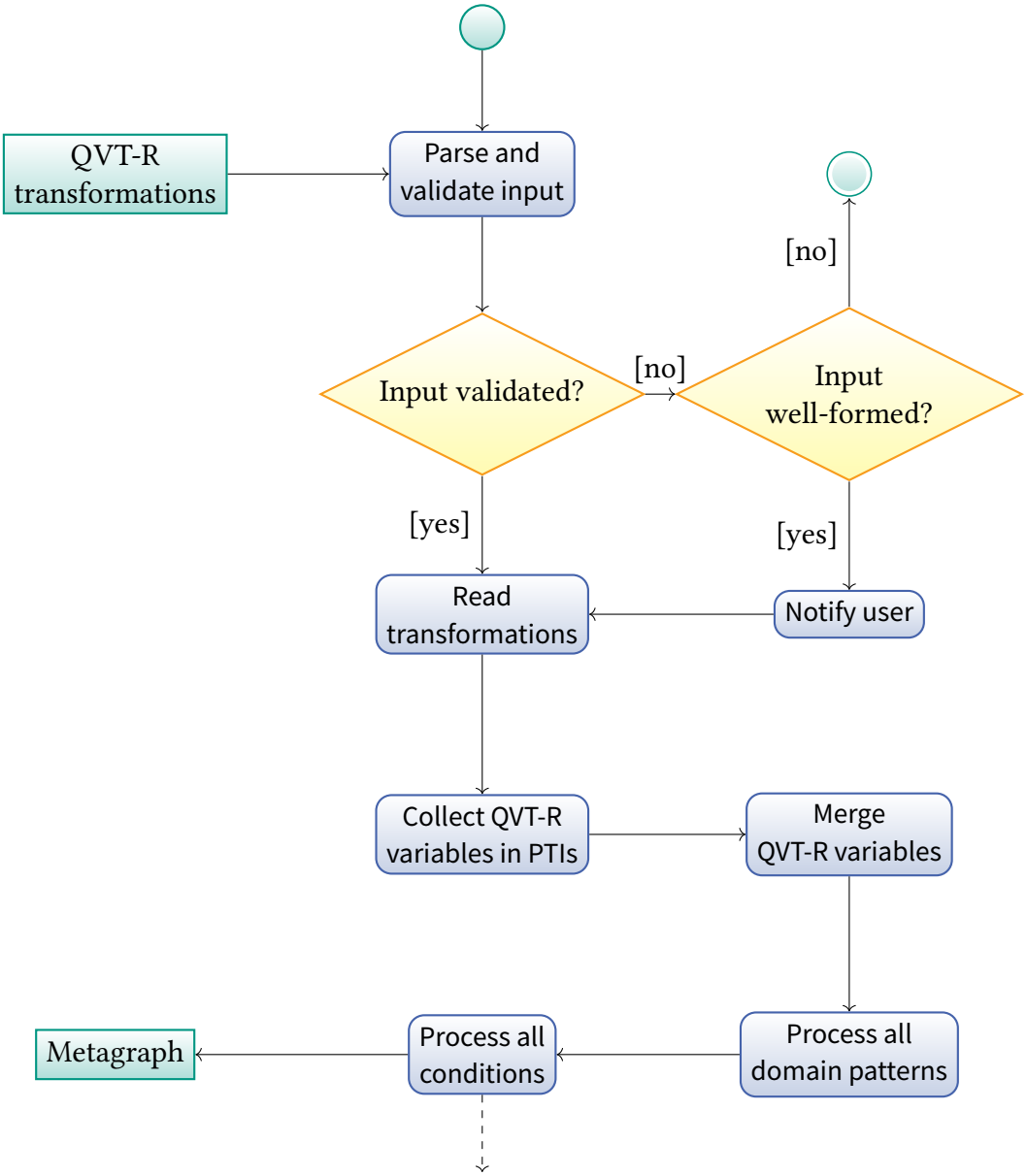


Figure 5.3

Figure 5.2.: Construction of the metagraph in the decomposition procedure

Figure 5.2

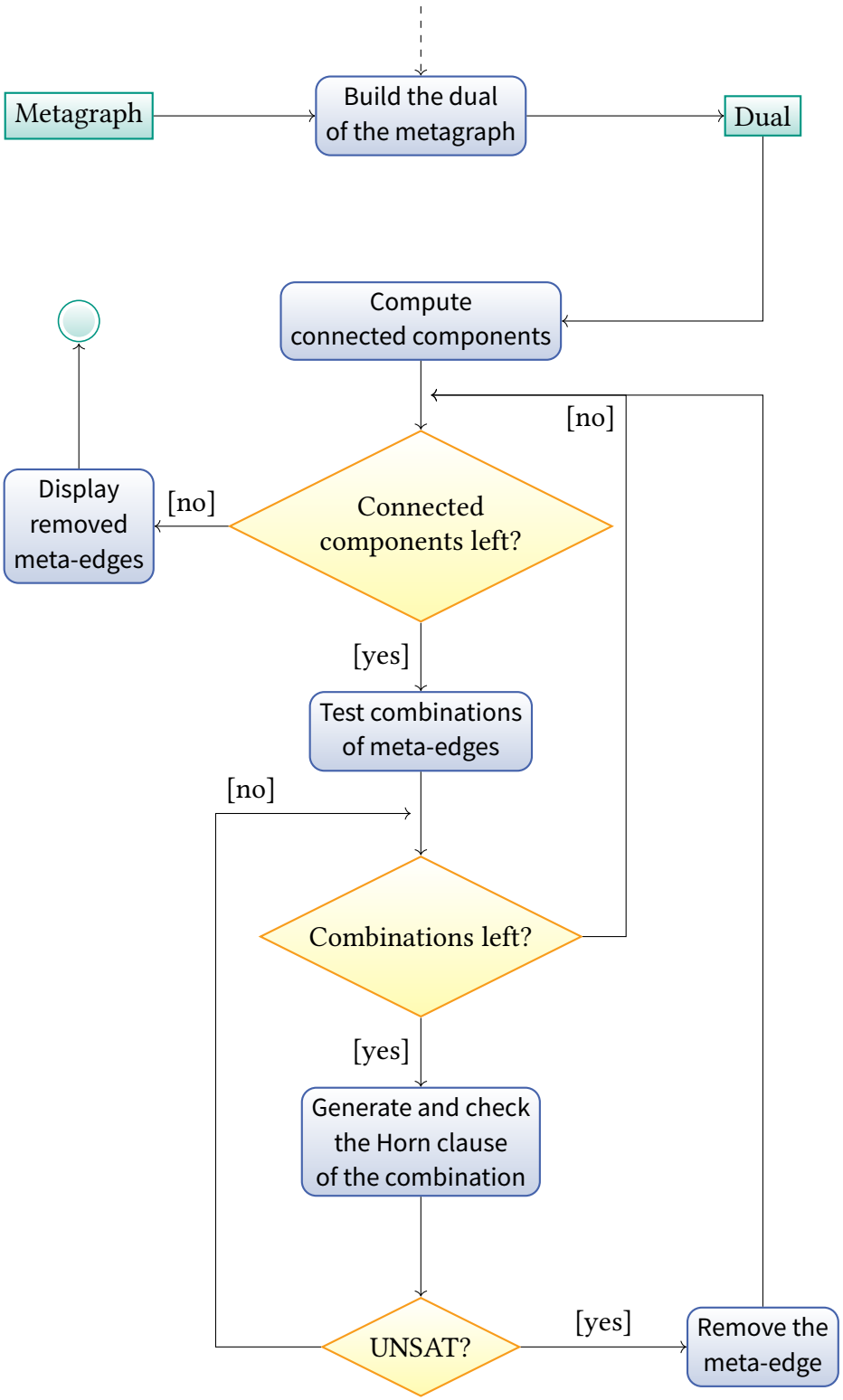


Figure 5.3.: Use of the metagraph in the decomposition procedure

5.3.1. Inputs of the Procedure

5.3.1.1. Read-Only Consistency Specifications

As stated in Section 4.1, the input of the decomposition procedure consists of two elements: a set of metamodels and a consistency specification on this set of metamodels. In this implementation, both elements are part a set of QVT-R files that serves as an input. An important point is that every access to either metamodels or transformations is read-only. This is consistent with the fact that the decomposition procedure does not update consistency specifications and that it does not perform consistency enforcement.

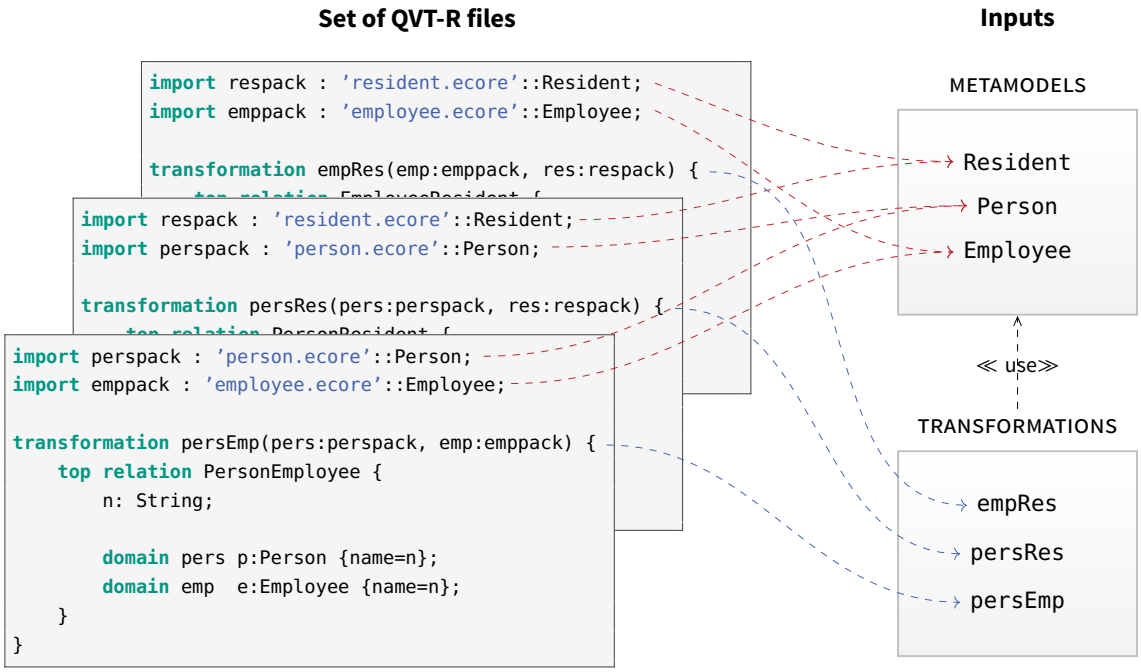


Figure 5.4.: From QVT-R files to metamodels and transformations

A QVT-R file is programmatically represented as a `Resource` object, which is the standard EMF mechanism for persistent documents. With an appropriate configuration based on dependency injection, resources are able to model contents of many languages and file formats, including OCL, QVT-R and Ecore.

Metamodels and transformations can then be retrieved directly from QVT-R files. If there are several files, they are processed fully and sequentially. First, metamodels of a file are retrieved thanks to **`import`** statements. Metamodels cannot be referenced without being imported. They are parsed at the same time as transformations. That is, the `Resource` gives access to **`import`** statements, which in turn give access to the roots of imported metamodels. In the context of QVT-R, roots of Ecore metamodels are packages (i.e. `EPackage` in Ecore). Once the root of the package is accessible, so is all its content.

Then, transformations are retrieved in the same way. Note that the `Resource` object first gives access to the concrete syntax tree (CST) of the QVT-R file, i.e. the syntactic structure

of the file according to the QVT-R grammar. While CSTs accurately reflect file contents, they do not offer a logical view of transformations. To meet this need, EMF provides an *ad hoc* binding called Pivot. The purpose of Pivot is to create a unified binding for representation of model-driven engineering languages in Java. Pivot offers good support of QVT-R, thus allowing to use QVT-R concepts in a natural way with Java classes.

The correspondence established by Pivot offers a set of classes similar to those of the metamodel represented in Figure 3.3.

If transformations are distributed over several QVT-R files, some metamodels may be used in multiple files, leading to a repetition of `import` statements. However, the input set of metamodels cannot include duplicates. For this reason, QVT-R files read with the same `ResourceSet` reference should import the same metamodel instances. This is consistent with the fact that metamodel elements appear at most once in the metagraph.

Figure 5.4 depicts the extraction of metamodels and transformations from QVT-R files. Emphasis is placed on the fact that each QVT-R transformation corresponds to a Java object, whereas metamodels that are imported many times result in a single object.

5.3.1.2. Validation of Input Specifications

There is no guarantee that input files – both QVT-R files and Ecore metamodels that these files reference – are valid. More precisely, three scenarios may arise:

1. Input files are *valid*. This is the correct and expected case. No error handling is required here. The consistency specification can be used to build the metagraph.

2. Input files are *well-formed but not valid*. This means that files are syntactically correct, i.e. they respect QVT-R and Ecore grammars, but there are semantic errors, e.g. two transformations with the same name or a domain pattern using a metaclass that does not exist.

3. Input files are *not well-formed*. This means that files are syntactically incorrect, i.e. they do not result in a consistency specification.

In terms of processing, there is a difference between the second and the third scenarios. In the second scenario, there exists a consistency specification although it may be incorrect. Given that the decomposition procedure manages consistency in metamodel instances expressed by consistency rules but not in consistency rules themselves, this is a non-blocking error. The causes of this error are displayed to the user but the procedure continues to run. In this case, the results of the procedure must be interpreted with caution. If there are semantic errors in the input specification, it is likely that there are also some in the output specification.

In the third scenario, however, the consequence of syntactic errors is that there is no usable specification. QVT-R files or metamodels with syntactic errors are useless, as transformations cannot be executed. For this reason, the decomposition procedure exits.

Another scenario, slightly different from those in the above list, relates to input files that are valid but useless for consistency preservation. For example, this scenario occurs with

a QVT-R relation whose precondition (**where** clause) always evaluates to FALSE. Regarding consistency, this means that the associated consistency rule is empty. As a result, all pairs of instances fulfill the consistency rule, i.e. all pairs of instances are consistent. Such a relation can be removed from a specification without altering consistency.

Error management is already part of the implementation of the QVT-R language. More precisely, errors can be accessed from the `Resource` object and they are represented as a list of `Resource.Diagnostic` objects, which is the standard way in EMF to report errors in persistent documents. This list can then be displayed in a user-friendly way.

5.3.2. Recursive Construction of QVT-R Concepts

At this stage, the decomposition procedure has well-formed input files forming a set of metamodels and a set of QVT-R transformations. The content of these transformations needs to be analyzed by the procedure in order to find metamodel elements and constraints that result in a metagraph. This subsection explains how to define a traversal order among and inside transformations.

5.3.2.1. Traversal Order Among Transformations

Transformations are the most outer objects of QVT-R files, in that there is no other object in the language that contain transformations. In other words, transformations are not subordinated to any other element and at the root of a QVT-R file, there are only transformations and imports required by these transformations. For these reasons, the transformations can be considered as independent of each other.

This property allow them to be grouped or distributed over multiple files. In the context of multi-model consistency preservation, this makes the development of independent consistency rules easier because the specification can be divided into as many files as necessary. It turns out that this is also of interest to the decomposition procedure.

Independence assures that the set of QVT-R transformations can be analyzed sequentially and in any order by the decomposition procedure. More precisely, a `Metagraph` object is instantiated and filled with the contents of transformations in turn so that all transformations are part of the same metagraph.

5.3.2.2. Traversal Order Inside Transformations

A QVT-R relational transformation consists in a set of relations and a set of keys. Inside a given transformation, there can be several relations participating in the definition of a consistency rule. Unlike transformations, relations cannot be processed in any order.

First, not all relations are top-level, i.e. not all relations are automatically invoked if their parent transformation is executed. To be executed, non-top-level relations must be explicitely called by other relations using **when** and **where** clauses. The process of relation invocation is explained in detail in Section 3.3.3.

The objective is to process relations in the order in which they would be invoked if the transformation was executed. There are two advantages to this. First, this ensures that dependencies resulting from parameters of relation invocations are respected. This way,

template expressions refer to variables which represent an already existing element in the metagraph. Then, relations that are both non-top-level and not invoked by any other relation are not processed by the decomposition procedure. By never being invoked, these relations are useless in the consistency specification.

To that purpose, we represent the set of relation invokations as a *call graph*. In general, a call graph models function calls in the control flow of a programm. A restriction is imposed on QVT-R specifications to make this representation easier to process for the procedure. From now on, we only allow invocations to appear in **`where`** clauses. Calling a relation *A* from the **`when`** clause of relation *B* is similar to calling *B* from the **`where`** clause of *A*. In the context of QVT-R, a call graph is a graph whose vertices are relations (of the same transformation) and edges are relation invocations.

The call graph is always a directed graph. Figure 5.5 gives an example of such a call graph for a transformation with six relations including two top-level relations. Numbers next to vertices indicate in which position the vertex was first visited. Note that the call graph may contain cycles, as nothing prevents circular references in QVT-R relations.

Starting from top-level relations results, relations can be processed using a depth-first traversal. This reproduces the order of invocation of relations if the parent transformations is executed. As a result, a relation is only visited if it is top-level or if a relation invoking it was itself visited before. This ensures that metamodel elements passed as parameters of the relation were already processed in another relation.

In addition to relations, transformations may contain keys. Keys in QVT-R are similar to keys in relational databases. They serve as unique identifiers for objects only used at

```
top relation A {
    where {B;}
}

relation B {
    where {C; D;}
}

relation C {}

relation D {
    where {A;}
}

top relation E {
    where {F;}
}

relation F {}
```

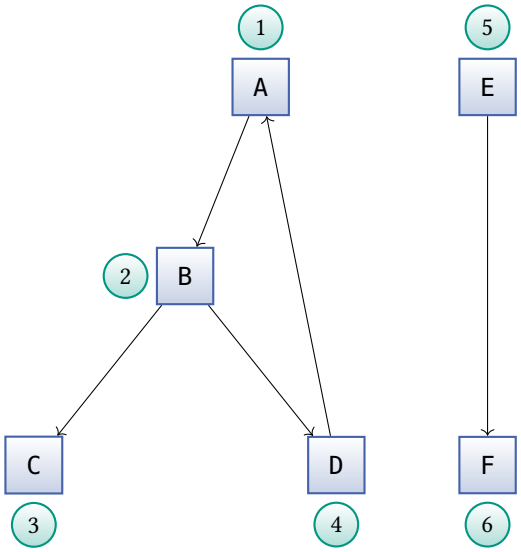


Figure 5.5.: Order of processing of QVT-R relations with invariants

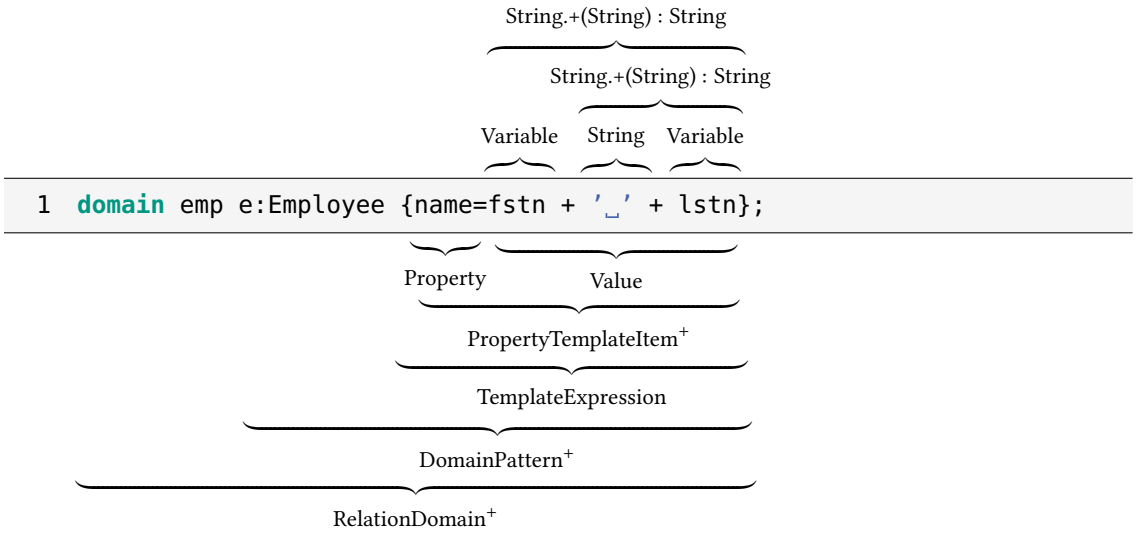


Figure 5.6.: Hierarchical structure of a QVT-R domain

the time of object creation. In case of consistency enforcement, i.e. when transformations are executed, they prevent the creation of duplicate objects. As they do not influence the declaration of consistency rules, they are ignored by the decomposition procedure.

By going through transformations and their relations according to the rules set out in the previous subsections, it is possible to process all relations while avoiding dependency problems. The next step is to fill the metagraph with the content of the relations.

5.3.3. Translation of Global Aspects of Specifications

At this stage, the traversal order of QVT-R relations is fully defined. This section shows how each relation can be processed in order to define new meta-vertices and meta-edges. This process is called *translation*.

The main idea behind the translation of relations is that metamodel elements are interrelated if they are bound to the same QVT-R variables. In QVT-R, regardless of the template expression, pattern matching exclusively relies on variables. Consequently, grouping metamodel elements according to the variables to which they are bound results in small sets of elements that define consistency rules. Such sets then form meta-edges.

Apart from this idea, some elements of relations remain to be translated. In particular, preconditions and invariants influence the application of relations and need to be integrated in the metagraph as well.

5.3.3.1. Finding Common Variables in Template Expressions

To group interrelated metamodel elements depending on variables they are bound to, it is necessary to traverse the syntax tree of template expressions, i.e. to explore OCL expressions in order to list these variables.

Figure 5.6 gives an overview of what a domain is made up of. This line of code is based on Example 5.1.1. The upper part of the figure depicts the subexpressions of the template expression in OCL. The lower part depicts the structure of the domain in QVT-R. The plus sign indicates that it is possible to concatenate several elements of the same type.

As shown on the figure, all QVT-R variables are located in the value of the property template item. We say that `name` is *bound* to the set of variables {`fstn`, `lstn`}. The best way to automatically retrieve variables from property template items is to use a visitor pattern. A visitor is a class that offers the possibility to define new operations on existing data structures without modifying them. As they belong to the visitor class, these operations are external to the data structures. More exactly, they are part of `visit` methods inside the visitor. The visitor pattern is a suitable solution when it comes to perform an analysis of abstract syntax tree.

The reason is that it allows the development of many different behaviours according to the type of the visited class. These behaviours take the form of methods in the body of the visitor class. First, the base class is designated as visitable — we say that it *accepts* a visitor, which is represented by a method *accept* with one argument, the visitor class. Then, each subclass overrides the *accept* method and calls one method of the visitor, sending itself (`this`) as an argument of the method so that the visitor can perform operations on the visited subclass.

In an abstract syntax tree, the base class is often an abstract `Expression` class. In OCL, this class is called `OCLExpression`. Then, variables, literals, operations, etc. of OCL are represented as subclasses of `OCLExpression`. Each subclass calls its own visitor method, e.g. `VariableExp` calls `visitVariableExp(this)` from the visitor class. By deriving the visitor class, a developer can write the logic of visitor methods and implement its own operations on visited objects without modifying these objects.

The extraction of QVT-R variables from OCL expressions can be achieved by using a visitor. The use of this pattern is beneficial here for two reasons. First, the OCL meta-metamodel is made accessible to the implementation of the decomposition procedure by the Pivot binding of OCL, a Java library. This binding already provides a visitor class to inherit for custom processing of expressions. Note that this visitor is itself included in a larger one that can also process elements of QVT-R, an `AbstractQVTrelationVisitor`.

Second and more importantly, the logic behind the extraction of these variables depends on the nature of the expression in the property template item. This need for special processing depending on OCL expressions motivates the use of a visitor.

Given the abstract syntax tree (AST) of an OCL expression, a node is called *terminal* if it is a leaf of the tree, i.e. if it contains no subexpression. Conversely, a node is called *non-terminal* if it is the root node or a branch node of the AST, i.e. if it is not a terminal node. Child nodes of non-terminal nodes are called *components*. In Figure 5.6, "`lstn`" is terminal whereas "`'␣' + lstn`" is non-terminal because it has two components.

A terminal node contains a variable if and only if it is a variable expression (`VariableExp`). A non-terminal node contains all the variables of its components. Therefore, extracting components can be achieved with a visitor as follows.

Each subclass `C` of `OCLExpression` corresponds to a `visitC` method in the visitor class. Each method of the visitor returns a set of variables. Let VARS be the function that takes

an OCL expression as an input and returns the set of variables in it. The function can be constructed inductively depending on the following cases.

- *(Base case)* For a terminal node that represents a variable expression, the visitor returns a unit set containing the variable in the expression.
  For example, VARS(`fstn`) = {`fstn`}.
- *(Base case)* For a terminal node that does not represent a variable expression, the method returns an empty set. For example, VARS(`1.5`) = {}.
- *(Inductive step)* For a non-terminal node, the method returns the union of sets of its components. The number and position of components depends on the node. For example, given three OCL expressions e_1, e_2 and e_3, VARS(e_1 `and` (e_2 `or` e_3)) = VARS(e_1) ∪ VARS(e_2 `or` e_3)) = VARS(e_1) ∪ VARS(e_2) ∪ VARS(e_3).

Using the visitor on the value of the property template item returns the set of variables to which the property is bound. After processing all property template items, each property that participates in the definition of a domain pattern is mapped to a (possibly empty) set of QVT-R variables.

5.3.3.2. Merge of Consistency Variables

Associating sets of variables with metamodel elements is useful to infer the structure of the consistency specification. If two metamodel elements share one or more variables (in the sense that the intersection of sets of QVT-R variables to which they are bound is non-empty), then the set of values that their instances can take is restricted. Instances must be consistent. Similarly, it is then possible to build sets of metamodel element, such that for every element in the set, there is another element which shares at least one QVT-R variable with it. Such a set is called form a meta-edge.

The operation described above is called the *merge of consistency variables* and implemented in Algorithm 5.1.

First, let $\{e_1, \ldots, e_n\}$ be a set of properties (metamodel elements) and let $\{v_1, \ldots, v_m\}$ be a set of QVT-R variables. In accordance with Section 5.3.3.1, each element e_i was mapped to a set $V_{e_i} \subseteq \mathcal{P}(\{v_1, \ldots, v_m\})$. As a result, the input of the algorithm is a set of pairs $\{(\{e_i\}, V_{e_i})\}$. The first element of the pair is a unit set containing the property e_i while the second element is the set V_{e_i} of QVT-R variables to which e_i is bound.

Similarly, the output of the algorithm is a set of pairs $\{(E_i, V_{E_i})\}$ where $E_i \subseteq \mathcal{P}(\{e_1, \ldots, e_n\}$ and $V_{E_i} \subseteq \mathcal{P}(\{e_1, \ldots, e_n\}$ such that the V_{E_i} sets are pairwise disjoint.

Each pair $(\{e_i\}, V_{e_i})$ is called an *entry*. The idea of the algorithm is to choose a reference entry for each loop in which other entries will be merged if they have at least one common variable with the reference. Merging two entries is achieved by merging their properties on one side and their variables on the other. If there was a merge at any point, this means that not all entries are pairwise disjoint and that the algorithm has to continue, hence the line 18 of the algorithm. Once that all entries have been compared with the reference

Algorithm 5.1 Merge of consistency variables

```
1  procedure Merge-Consistency-Variables({(e_i, V_{e_i})})
2    stopMerge ← True
3    entries ← [({e_i}, V_{e_i})]
4
5    do
6      stopMerge ← True
7      results ← {}
8
9      while not entries.isEmpty() do
10       ref = ({e_ref}, V_{e_ref}) ← entries[0]
11       others ← entries[1:]
12       entries ← []
13
14       for ({e}, V_e) :  others do
15         if V_e ∩ V_{e_ref} = ∅ then
16           entries.add(({e}, V_e))
17         else
18           stopMerge ← False
19           ref ← ({e} ∪ {e_ref}, V_e ∪ V_{e_ref})
20         end if
21       end for
22       results.add(ref)
23     end while
24     entries = results
25   while not stopMerge
26
27   return set(entries)
28 end procedure
```

entry, another entry is chosen as a reference among those that could not be merged with the previous reference. This is repeated until there are no more entries to process. As a result, all entries have been either chosen as reference entries or merged with other entries when the algorithm terminates.

Example 5.3.1 *Let $\{e_1, \ldots, e_4\}$ be a set of metamodel elements and let $\{a, b, c, d, e\}$ be a set of QVT-R variables to bind these elements. Figure 5.7 shows the execution of Algorithm 5.1 on the following input:* $(\{e_1\}, \{a, b\}), (\{e_2\}, \{c, d\}), (\{e_3\}, \{e\}), (\{e_4\}, \{b, d\})$.

In Figure 5.7, entries depicted in green are part of the `results` *set. in Each pass corresponds to a whole execution of the* `while` *loop in the algorithm, i.e. until* `entries` *is empty. At the end of the pass, there is two possibilities, depending on whether* `stopMerge` *was assigned to* False *during the* `while`*. If so, there was a merge in the previous pass so there may still be others. In the example, it is the case for the first and the second passes. In this case, the*

temporary results become `entries` *again and a new pass is initiated. In the third and last pass, there is no merge so* `stopMerge` *remains* TRUE. *Results are now final.*

After the merge of consistency variables, metamodel elements that occur as properties in a QVT-R relation are grouped according to variables they share. If metamodel elements form vertices of an hypergraph, these sets of metamodel elements can be considered as hyperedges. Thus, the global aspect of consistency specifications (i.e. consistency relations) can be reproduced at the metamodel element level.

5.3.4. Translation of Local Aspects of Specifications

To fulfill the definition of a metagraph, it is still necessary to associate with each hyperedge a set of consistency rules. The reason for this is that the existence of a group of metamodel elements only indicate that instances of these elements must be consistent but it does not specify under which rules they should be.

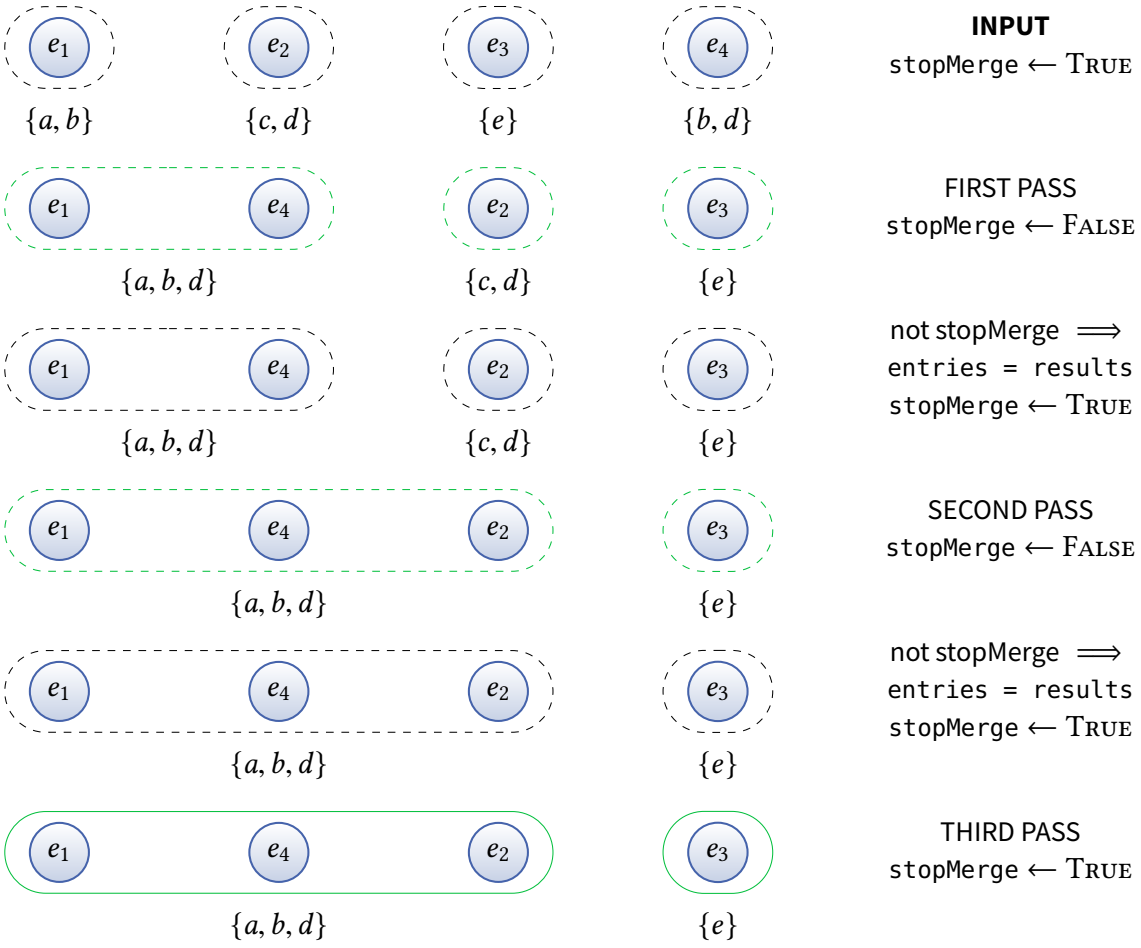


Figure 5.7.: Example of a merge of consistency variables

Figure 5.8 shows an example in which two relations using the same metaclasses and the same properties result in the same hyperedges after the execution of Algorithm 5.1, although they express different consistency specifications.

```
top relation ResidentEmployee {
    fstn: String;
    lstn: String;

    domain pers r:Resident {firstname=fstn, lastname=lstn};
    domain emp e:Employee {name=fstn + '␣' + lstn};
}
```

```
top relation ResidentEmployee {
    fstn: String;
    lstn: String;

    domain pers r:Resident {firstname=fstn, lastname=lstn};
    domain emp e:Employee {name=lstn.toUpper() + ',␣' + fstn};
}
```

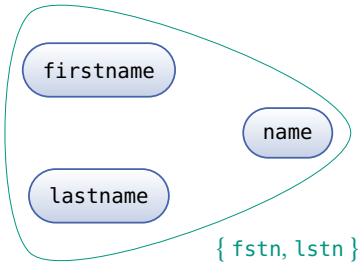


Figure 5.8.: Two consistency specifications resulting in the same hyperedge

In order to model the consistency specification in its entirety at the metamodel element level, this subsection details the implementation choices of the procedure and the reasons for these choices. First, we explain the chosen formalism to embed consistency rules into hyperedges. Then, we look at how to translate contents of domain patterns. Finally, we consider other elements from QVT-R relations, e.g. preconditions and invariants, that must be translated as well.

5.3.4.1. Symbolic Representation of Consistency Rules

The integration of consistency rules into the metagraph has one major purpose, the detection of redundancy in the specification. In particular, we aim to find an algorithmic approach to implement decomposition methods described in Section 4.2. The point of consistency rules in these methods is that they give conditions for instances of a set of metamodel elements to be consistent with each other.

Sets of instances defined by consistency rules and used in Definitions 4.2.3 and 4.2.6 may be possibly infinite. This case occurs very often in practice. For example, a consistency rule that ensures that two `String` attributes have the same value generates an infinite set of consistent instances, given that there are infinitely many strings. This makes it impossible to compare element by element, i.e. extensionally, the set of instances that fulfill a consistency relation with the set of instances that fulfill an alternative combination of consistency relations.

Consequently, an implementation of decomposition methods must compare sets intensionally, i.e. by comparing their internal definitions. This is actually consistent with the fact that consistent instances of a metamodel are never defined extensionally. When using

QVT-R to specify consistency, definitions of sets of consistent instances are ultimately written as OCL expressions in domain patterns, preconditions and invariants.

Example 5.3.2 *In order to show how redundancy can be identified from definitions of consistency rules, let* MeterValue, InchValue *and* FootValue *be three metaclasses of three metamodels representing units of length. Each metaclass has an attribute* value *of type* `Real`.

A consistency specification is associated with these metamodels. Here, consistency ensures that if there exists an instance of one of these metaclasses, then there exist instances of other metaclasses such that conversions between length units are correct. For example, if there is an instance of FootValue *with* value `1`, *there must be an instance of* InchValue *with* value `12` *and an instance of* MeterValue *with* value `0.3048`.

This consistency specification can also be written with OCL expressions as follows:

(1) `InchValue.value = 12 * FootValue.value`

(2) `MeterValue.value = 0.0254 * InchValue.value`

(3) `MeterValue.value = 0.3048 * FootValue.value`

In terms of consistency relations, this specification with three metamodels and three consistency rules form a triangle graph. Using a decomposition method, we want to check if one of these relations is redundant, e.g. rule (3). Therefore, we find a combination of consistency relations with the same endpoints, here (1)-(2).

According to Definition 4.2.3, (3) is totally redundant if it has the same set of consistent instances as the combination (1)-(2). In this example, such a set is infinite and cannot be fully generated: each value *attribute can represent infinitely many lengths. An exhaustive comparison of sets is then impossible. A proof of redundancy cannot be obtained extensionally.*

It is however clear that two rules are sufficient to deduce a third one. If models fulfill rules (1) and (2), then they fulfill the following rule deduced from (1) and (2):

$$\begin{aligned} &\texttt{MeterValue.value = 0.0254 * (12 * FootValue.value)} \\ \Longleftrightarrow\ &\texttt{MeterValue.value = 0.3048 * FootValue.value} \iff (3) \end{aligned}$$

Regarding sets of consistent instances, an instance that is consistent according to the combination (1)-(2) is also consistent according to (3) given the result above. As a result, (3) represents a redundant consistency relation. This proof of redundancy was obtained intensionally, i.e. only by using definitions of consistency rules.

As shown in Example 5.3.2, the solution for an implementation of decomposition methods lies in the intensional comparison of sets of instances, i.e. in the manipulation of the definitions of these sets. In other words, OCL expressions (and more generally consistency rules in QVT-R) that define consistency rules are regarded as sets of symbols that can be manipulated to infer certain results. This is a case of *symbolic computation*.

Thanks to the static analysis of consistency specifications, symbols in OCL and QVT-R expressions are already defined. They correspond to subexpressions or language constructs,

i.e. nodes of the abstract syntax tree of the specification. For example, Figure 5.6 depicts the structure of a domain pattern in QVT-R. All elements highlighted in this structure can be used to identify sources of redundancy, as in the previous example.

Therefore, an emerging approach for an implementation of decomposition methods in this work involves the use of tools for symbolic computation during the static analysis of QVT-R files. Reasoning about OCL expressions is not trivial, as it is an expressive language with a quite large syntax. For example, a tool for symbolic computation that works for OCL should be able to deduce that the expressions "`firstname + '␣'+ lastname`" and "`firstname + '␣'+ ''+ lastname`" are equivalent without assigning values to `firstname` and `lastname` because everything in these expressions is a symbol, not a variable.

In practice, it is unusual to write a tool for symbolic computation that only works for one language due to the complexity of the task if the language is not trivial. Most often, these tools have their own formalism. They provide description of expressions and elements on which they are able to reason and give results. Chapter 6 provides an explanation as for the choice of an appropriate tool for the decomposition procedure. It also describes how and to what extent OCL expressions and QVT-R constructs can be translated as formulae interpretable by this tool.

At this stage, the approach is as follows. First, definitions of consistency rules within QVT-R relations are translated into expressions of a symbolic and formal language, so that there exist algorithms to deduce redundancy results from these definitions. Then, these expressions are embedded in the hypergraph of Section 5.3.3. This completes the construction of the metagraph. It therefore remains to be determined which parts of QVT-R relations have to be embedded and how.

5.3.4.2. Processing of Domain Patterns

Definitions of consistency rules mainly depend on domain patterns, i.e. on contents of template expressions. In particular, metamodel elements are primarily bound to QVT-R variables in template expressions. In accordance with the definition of the metagraph, the idea is to associate this definition with a set of metamodel elements, i.e. a hyperedge, so that all the information necessary to express consistency for this hyperedge can be accessed from the hyperedge.

For a set of instances – whose consistency is determined by a set of property template items – to be consistent, there must exist an assignment to QVT-R variables so that all property template items evaluate to TRUE. Property template items bind a metamodel element (*property*) with an OCL expression (*value*). When they are combined, they can also be regarded as systems of equations. If so, metamodel elements are replaced with the values of their instances. Instances are then consistent if the system has a solution.

Example 5.3.3 *The QVT-R relation in Listing 5.2 involves three property template items (PTIs). The = sign in PTIs is an assignment operator. Variables* `fstn` *and* `lstn` *take some arbitrary values and are then assigned to attributes of metaclasses through PTIs. Resulting instances are consistent because they were built to fulfill domain patterns.*

We now represent an instance as a tuple (`firstname`, `lastname`, `name`) *that is as an element of String* × *String* × *String. Consistency of a given set of instances can be determined by modeling PTIs as a system of equations. For* (`'Alice'`, `'Smith'`, `'Alice␣Smith'`)*:*

$$\begin{cases} \texttt{firstname=fstn} \\ \texttt{lastname=lstn} \\ \texttt{name=fstn + '␣'+ lstn} \end{cases} \iff \begin{cases} \texttt{'Alice'=fstn} \\ \texttt{'Smith'=lstn} \\ \texttt{'Alice␣Smith'=fstn + '␣'+ lstn} \end{cases}$$

Given that the system has a solution, these models are consistent. The point of this representation is that consistency fulfillment can be interpreted as the solving of a system of equations. The manipulation of equations is an important feature of tools for symbolic computation.

For this reason, the consistency rule associated with a hyperedge is the conjunction of all property template items whose property is in the hyperedge. These property template items are interpreted as equations, meaning that the consistency rule is interpreted as a system of equations. Accordingly, consistency fulfillment in a set of hyperedges can be interpreted as a system made up of equations of all hyperedges.

The approach presented in Example 5.3.3, i.e. replacing metamodel elements with values of their instances and checking if the system of equations has a solution, can then be adapted to the decomposition procedure. As for the combinations of consistency relations, let E be a meta-edge and $\underline{E}$ a combination of meta-edges such that all meta-vertices in E are in the first or in the last meta-edges of $\underline{E}$. If when the system of equations induced by E is solvable, so is the system induced by $\underline{E}$, then all instances consistent according to E are also consistent according to $\underline{E}$. This means that E is totally redundant.

In other words, reasoning about domain patterns in terms of systems of equations is a way to implement the detection of redundancy as exposed in the decomposition methods. As a result, property template items are not embedded in metagraphs as OCL expressions. They are first translated as equations in an appropriate language that tools for symbolic computation are able to use.

In the decomposition procedure, the translation is based on a visitor again, i.e. the same mechanism as the search for common variables in domain patterns. The abstract syntax tree is explored thanks to a visitor class, each node being translated in a specific way. The translation is described in a dedicated chapter on constraint translation, Chapter 6.

```
1     top relation ResidentEmployee {
2         fstn: String;
3         lstn: String;
4
5         domain pers r:Resident {firstname=fstn, lastname=lstn};
6         domain emp e:Employee {name=fstn + '␣' + lstn};
7     }
```

Listing 5.2: Processing of domain patterns for Example 5.3.3

5.3.4.3. Processing of Preconditions and Invariants

In addition to property template items, there are three QVT-R constructs in which OCL expressions can occur. These are preconditions (in `when` clauses), invariants (in `where` clauses) and local invariants (after the template expression of a domain pattern).

These constructs also contribute to consistency specification. As a result, the translation of template expressions in invariants and preconditions is achieved using the same visitor as the translation of property template items.

In the context of this thesis, OCL expressions apart from template expressions in domain patterns are restricted to the manipulation of QVT-R variables. Using root variables, it is possible to reference metamodel elements directly in local invariants, `when` and `where` clauses. Since this makes the generation of the metagraph harder for a low gain in expressiveness – domain patterns already allow to do this in an organized way, this is not supported in the decomposition procedure.

Their integration into decomposition methods is presented in Section 5.4.4.

5.4. From Metagraph To Decomposition

The second part of the decomposition procedure relates to the use of the metagraph in order to find independent subsets of consistency relations. In this section, an implementation of decomposition methods discussed in Section 4.2 is presented.

5.4.1. Metagraph Dual

5.4.1.1. Construction of the Dual

Being based on hypergraphs, metagraphs have the advantage of being very expressive when it comes to model relations with variable arities. The downside is that common graph algorithms (graph traversal, connected components, etc.) become harder to define, to analyze and to apply. The choice between graphs and hypergraphs is then a balance between abstraction and usability.

This problem also occurs in the field of constraint satisfaction when problems are modeled with constraint hypergraphs. A common approach is to transform the hypergraph into an isomorphic graph that allows to compute the same solutions, the *dual of the constraint hypergraph* [DC+03]. Definition 5.4.1 adapts the concept of dual to metagraphs.

Definition 5.4.1 *Let $\mathfrak{M} = (\mathcal{H}, c)$ be a metagraph defined for a consistency specification $\underline{\mathcal{R}}$. The* **dual of the metagraph** $\mathfrak{M}$*, denoted $\mathfrak{M}^*$, is a tuple $(\mathcal{G}, v, c)$ with a simple graph $\mathcal{G}$ and two functions, v and c such that:*

- $V_{\mathcal{G}} = E_{\mathcal{H}}$
- $E_{\mathcal{G}} = \{\{E_1, E_2\} \mid \forall\, (E_1, E_2) \in E^2_{\mathcal{H}} : E_1 \cap E_2 \neq \emptyset\}$
- $\forall\, \{E_1, E_2\} \in E_{\mathcal{G}} : v(\{E_1, E_2\}) = E_1 \cap E_2$

Meta-edges of the metagraph become vertices of the dual. If meta-edges share at least one metamodel element, their corresponding vertices in the dual are linked. The function v is an edge labeling that indicates which metamodel elements two vertices of $\mathcal{G}$, i.e. two hyperedges of $E_{\mathcal{H}}$ have in common. In order to keep a representation that can be translated in both directions, the function c' is the same as the function c and associates a set of consistency rules with a set of metamodel elements.

For example, Figure 5.9 shows a metagraph for a consistency specification with three metamodels and its dual.

The dual of the metagraph is a simple graph, meaning that algorithms such as the computation of connected components or the search of paths become applicable. Building the metagraph can be considered as a pre-processing for the decision part of the decomposition procedure. Moreover, the dual contains all the information necessary to build the metagraph again. Given a dual $\mathfrak{M}^* = (\mathcal{G}, v, c')$, the metagraph $\mathfrak{M} = (\mathcal{H}, c)$ can be built by considering $V_{\mathcal{H}} = \bigcup_{V \in V_{\mathcal{G}}} V$, $E_{\mathcal{H}} = V_{\mathcal{G}}$ and $c = c'$.

5.4.1.2. Characterization of Combinations in the Dual

In a consistency relation graph, a combination of consistency relations has been defined as a path in the graph. At the metamodel element level, a combination in a metagraph has been defined as a sequence of meta-edges in which two consecutive terms share at least one meta-vertex, i.e. they have at least one metamodel element in common. Whatever the formalism, the aim of combinations is to represent an alternative sequence of consistency rules to detect if a consistency rule carries redundant information.

An equivalent characterization of combinations can be defined in duals of metagraphs. To check if the meta-edge 3 of the metagraph in Figure 5.9 is redundant, a combination to try could be 1–2. Valid combinations are those whose endpoints share at least one vertex (or meta-vertex) with the edge (or meta-edge) analyzed. Detecting candidate endpoints is trivial in the dual of the metagraph: it is the set of neighbours of the vertex representing the meta-edge. If two meta-edges share a meta-vertex in the metagraph, their vertex representations are linked by an edge in the dual.

As a result, valid combinations in the dual graph are paths that start with the rule to analyze and end with the rule to analyze. In other words, valid combinations are cycles in the dual graph. For a given cycle, consistency rules that participate in the redundancy checking are those associated with vertices in the cycle (except the one to be anayzed). The cycle must be simple, i.e. it does not go through the same vertex twice (except for the vertex to be analyzed which is both the first and the last vertices of the cycle).

In the context of the decomposition procedure, some cycles are more interesting than others. The difference between two cycles stems from metamodel elements shared between their vertices – this is provided by the function v of the dual. The set of metamodel elements of a cycle is the union of sets of metamodel elements on edges of the cycle, i.e. $\bigcup_{E \in C} v(E)$ for a cycle C. The more metamodel elements a cycle contain, the more information it has to detect redundancy. More importantly, it is most of the time necessary that all metamodel elements of the meta-edge to analyze are also part of the metamodel elements of the cycle.

On that basis, a first heuristic to find good candidate combinations (i.e. cycles in duals) is to favour those that contain all elements of the consistency rule to analyze. Another idea is to favour shorter cycles. Shorter cycles contain less consistency rules, meaning that verifications are easier to perform for the symbolic computation tool. Morever, two metamodels are more likely to be interrelated if they share many consistency rules. Following that logic, it is more appropriate to test shorter cycles, i.e. cycles made up of metamodels closely related to each other, before long cycles.

5.4.2. Independent Subsets of Meta-Edges

The first decomposition method presented in Section 4.2 uses the absence of consistency relations between sets of metamodels as an argument of independency between these sets. This section describes a similar result at the metamodel element level and a way to make use of this result in the decomposition procedure.

5.4.2.1. Connected Components in Metagraphs

The definition of connected components can be extended to hypergraphs and therefore to metagraphs. The aim is to describe independent sets of meta-edges. As a reminder, meta-edges are based on hyperedges, which are sets of vertices of the hypergraph.

Definition 5.4.2 *Let $\mathfrak{M} = (\mathcal{H}, c)$ be a metagraph. A* **subhypergraph** *of $\mathfrak{M}$ induced by a set of hyperedges $\mathcal{A}$ is a couple $\mathcal{S}_{\mathcal{A}} = (V, \mathcal{A})$ such that $\mathcal{A} \subseteq E_{\mathcal{H}}$ and $V = \bigcup_{A \in \mathcal{A}} A$.*

Subhypergraphs are a generalization of subgraphs for hypergraphs. Given that the cardinality of hyperedges is not fixed, there are several ways to define this concept. The one defined above is the most straightforward. It consists in selecting a subset of hyperedges and all vertices in these hyperedges. The fact that it is defined from hyperedges stems from the idea that meta-edges are the main elements of the decomposition procedure (e.g. to find combinations and to keep track of consistency rules).

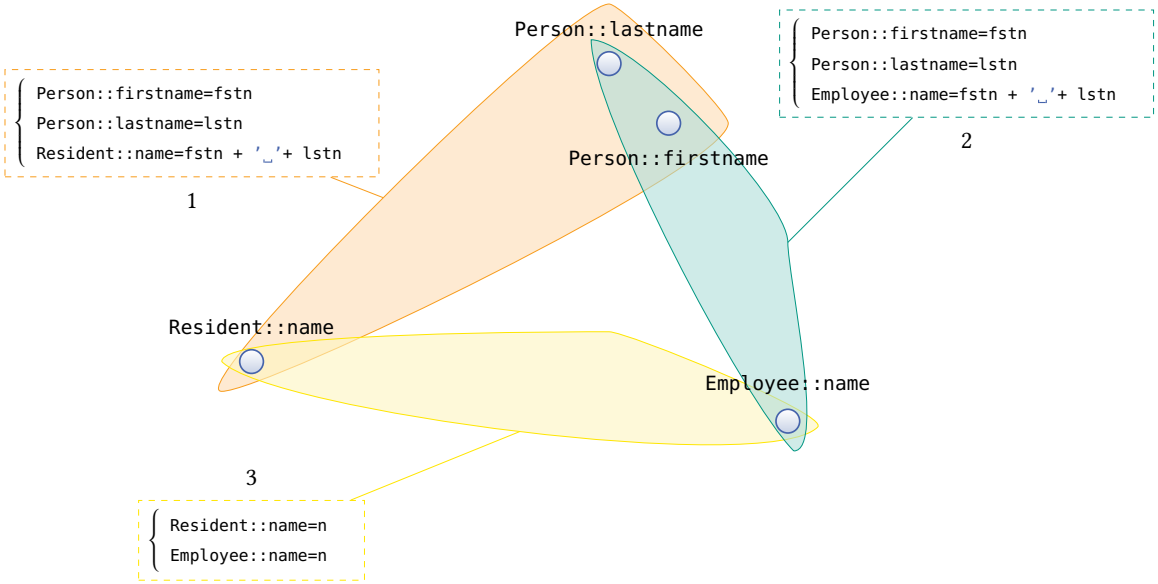
Definition 5.4.3 *Let $\mathfrak{M} = (\mathcal{H}, c)$ be a metagraph defined for a consistency specification $\underline{\mathcal{R}}$. A subhypergraph $\mathcal{S}_{\mathcal{A}}$ of $\mathfrak{M}$ induced by a set of hyperedges $\mathcal{A}$ is a* **connected component** *of $\mathfrak{M}$ under the following conditions:*

(1) $\forall\, v \in V_{\mathcal{S}_{\mathcal{A}}}, \forall\, E \in (E_{\mathcal{H}} \setminus \mathcal{A}) \;:\; v \notin E$

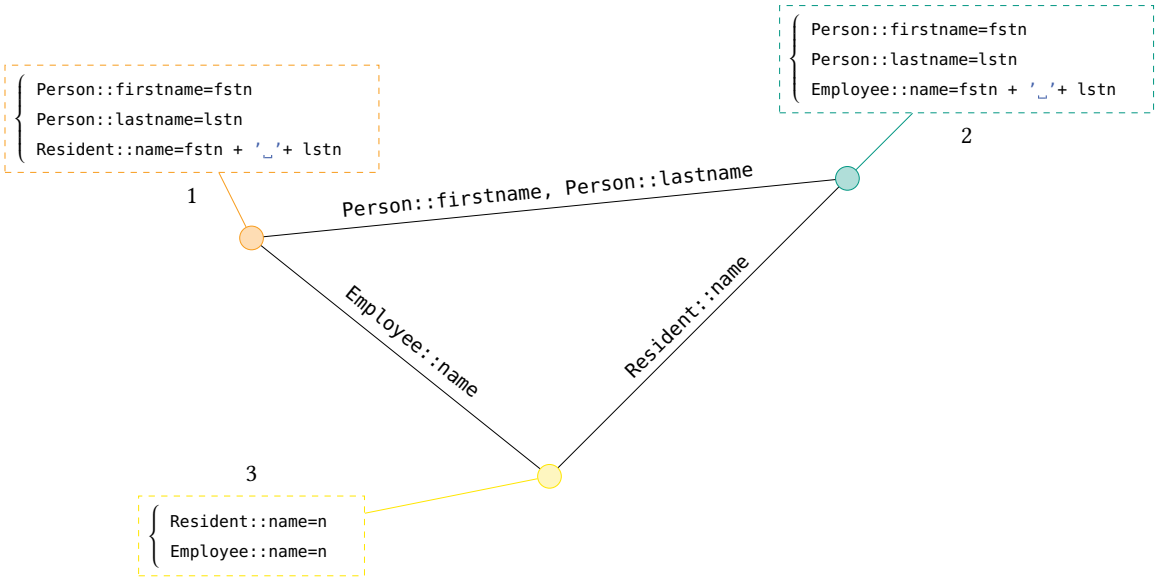
(2) $\forall\, u, v \in V^2_{\mathcal{S}_{\mathcal{A}}}$: *u is reachable from v*

In other words, a subhypergraph is a connected component if each of its vertices has the same degree in the hypergraph $\mathcal{H}$ and in the subhypergraph $\mathcal{S}_{\mathcal{A}}$ and if there exists a path between any two of its vertices.

In terms of consistency, two meta-edges have to be in the same connected component for one to be part of a combination to replace the other. Therefore, connected components are independent subsets of meta-edges. As for duals, a metagraph and its dual lead to same connected components. Two meta-edges are in the same connected component in the metagraph if and only if there is a path between the vertices they represent in the dual.



(a) Metagraph



(b) Dual of the metagraph

Figure 5.9.: A metagraph of a consistency specification and its dual

5.4.2.2. Computation of Connected Components

Given that duals are simple graphs, there are fast and efficient algorithms for the computation of connected components. The one used in the implementation of the decomposition procedure was first described by Hopcroft and Tarjan [HT73]. It runs in linear time using a recursive depth-first search.

5.4.2.3. Consequences of Independent Subsets in Metagraphs

At this stage, connected components denote two different properties of consistency preservation depending on the structure in which they are used. At the metamodel level, i.e. in consistency relation graphs, two consistency relations in separate connected components are independent in the sense that metamodels at their respective endpoints are not interrelated. At the metamodel element level, i.e. in metagraphs, two meta-edges in separate connected components are independent in the sense that one cannot be used in a combination to test if the other is redundant.

The existence of connected components in metagraphs is a stronger result than the existence of connected components in consistency relation graphs. The reason for this is that if two consistency relations are in separate connected components, then their respective meta-edges are separate too and cannot lie in the same connected component.

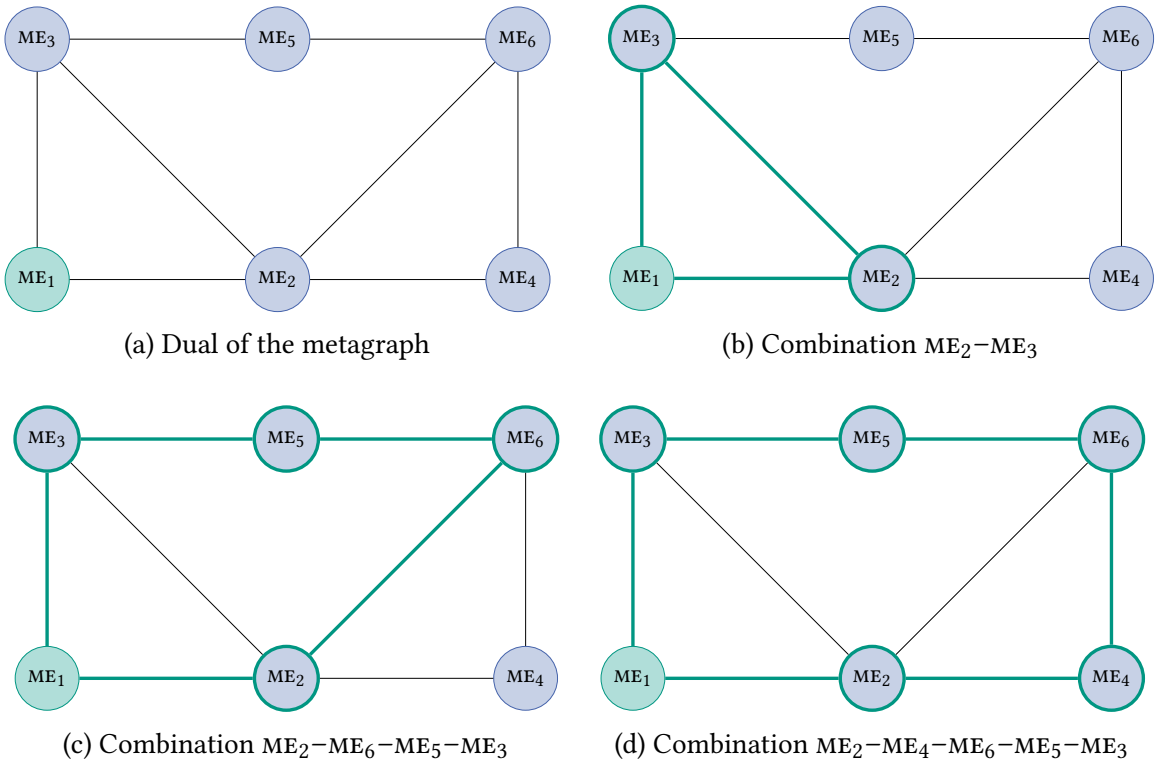
However, the converse does not hold. That is, two meta-edges in different connected components can be part of the same consistency relation. For example, let `A` and `B` be two metamodels with two metaclasses each, (`A1`, `A2` and `B1`, `B2`). Let the consistency specification on $\{\texttt{A}, \texttt{B}\}$ ensure that `A1` models are consistent with `B1` models and that `A2` models are consistent with `B2` models. In the resulting metagraph, there will be two connected components: one with the properties of `A1` and `B1`, another with the properties of `A2` and `B2`. In other words, although `A1` and `A2` are part of the same metamodel, they appear in separate connected components.

This is of interest for the implementation of the decomposition procedure. This means that consistency of $\{\texttt{A1}, \texttt{B1}\}$ and of $\{\texttt{A2}, \texttt{B2}\}$ can be verified separately. If we now assume that we can replace the meta-edge made up of elements of `A1` and `B1` with a combination of meta-edges from other metamodels, then $\{\texttt{A1}, \texttt{B1}\}$ is redundant whereas $\{\texttt{A2}, \texttt{B2}\}$ is not. Thus, the consistency specification between `A` and `B` is *partially redundant*.

5.4.2.4. Use of Connected Components in Metagraphs

The implementation of decomposition methods using the metagraph can finally be described as follows. First, connected components of the metagraph are computed. Then, for each connected component, as much meta-edges as possible are removed by verifying if they can be replaced by a combination of meta-edges. Candidate combinations must fulfill conditions explained in Section 5.4.1.2. Two cases are then possible. If all meta-edges associated with a consistency relation are ultimately removed, the relation is said to be *totally redundant*. Otherwise, the relation is said to be *partially redundant*.

It remains to be seen how alternative combinations of meta-edges can be generated in an independent subhypergraph and how they can be compared to meta-edges to analyze in order to detect redundancy. This is the purpose of the following sections.



(a) Dual of the metagraph

(b) Combination ME_2–ME_3

(c) Combination ME_2–ME_6–ME_5–ME_3

(d) Combination ME_2–ME_4–ME_6–ME_5–ME_3

Figure 5.10.: Three valid combinations to replace a meta-edge

5.4.3. Generation of Combinations of Meta-Edges

In the dual of a metagraph, an alternative combination of meta-edges is represented as a *simple* cycle with the additional condition that the meta-edge to analyze must be part of the cycle. In Figure 5.10, the dual of a metagraph with six meta-edges is depicted. If the aim is to remove ME_1, there are three valid cycles, i.e. three valid combinations to test.

Cycles sought in the context of the decomposition procedure must be simple. The reason for this is that cycles of meta-edges (i.e. vertices in the metagraph dual) with repeated vertices are superfluous in terms of consistency. The role of a meta-edge is to select consistent instances among all possible instances. When repeated, a meta-edge does not select more consistent instances than before.

Also, if a graph has at least one cycle, then it has infinitely many of them. Following the requirements of Section 5.4.1.2, this section describes an algorithm to find simple cycles in the dual of the metagraph, leading to the generation of a set of alternative combinations of meta-edges.

5.4.3.1. Cycle Enumeration

The problem of finding all simple cycles in an undirected graph is called *cycle enumeration*. Before presenting an appropriate algorithm to solve this problem in the context of the decomposition procedure, there are important details to mention. First, the number of cycles can be exponential in the number of vertices of the graph. This is especially the

case of the complete graph (since there are already $n!$ permutations of a set of n vertices). As a result, there is no polynomial algorithm to enumerate cycles in an undirected graph.

In the same way, this result is also valid for the enumeration of combinations in the metagraph. There can be exponentially many paths between two meta-edges in the metagraph. Of the two representations, duals are preferred to find combinations. That is, combinations are generated from cycles in the dual rather than from paths in the metagraph. This avoids the need to manage fine details of hypergraphs in the implementation.

The previous result can be relativized. Once a suitable combination (i.e. a combination that proves redundancy) is found, there is no need to enumerate other combinations. For example in Figure 5.10, if ME_1 and the combination ME_2–ME_3 are redundant, there is no need to find and to test other combinations. Also, as explained in Section 5.4.1.2, it is more likely to find redundancy in short cycles, i.e. with close meta-edges.

There are generally two families of algorithms to enumerate simple cycles in undirected graphs: those using *cycle bases* and those using *search algorithms* [MD76]. In this implementation, the former is preferred. Search algorithms being based on directed graphs, their use on undirected graphs requires to transform them into directed graphs by doubling every edge. Regarding the decomposition procedure, this would introduce an additional graph formalism, i.e. an additional layer of complexity.

5.4.3.2. Cycle Bases For Cycle Enumeration

A *cycle basis* in an undirected graph is a set of simple cycles which can be combined to generate all other simple cycles of the graph. For example, Figure 5.11 depicts a cycle basis for the dual of Figure 5.10. The basis is made up of three cycles.

The generic way to compute a cycle basis is to compute the spanning tree of the graph. Then, edges that are not part of the spanning tree are visited one by one and combined with some edges of the tree to form a simple cycle. The efficiency of the algorithm then depends on the choice of these edges. A well-known algorithm to find a cycle basis in an undirected graph is Paton's algorithm [Pat69]. This is the one used in the implementation of the decomposition procedure.

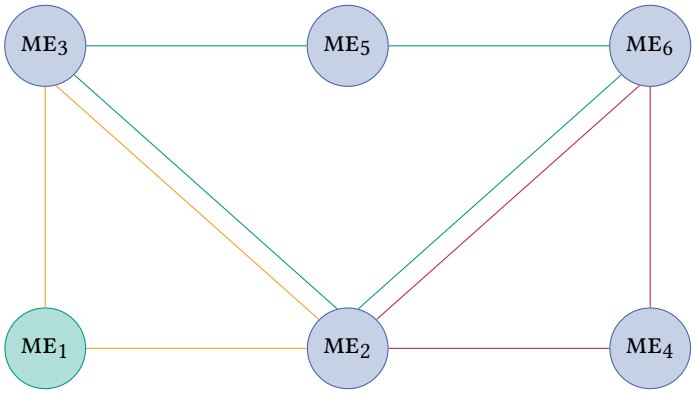


Figure 5.11.: Cycle basis for the dual of Figure 5.10

The next step is to use the cycle basis to enumerate all simple cycles. In order to do this, two or more cycles are merged at each enumeration. Not all merges produce cycles. In particular, cycles must share some edges. In the context of the decomposition procedure, there is an additional requirement: a cycle must go through the meta-edge to analyze in order to produce a valid combination.

Algorithm 5.2 is a slightly modified version of Gibb's algorithm to enumerate simple cycles in an undirected graph using a cycle basis [Gib69]. In this algorithm, every cycle is represented as a set of edges. The $\oplus$ sign denotes the symmetric difference, i.e. $A \oplus B$ is the set of edges that are in A or in B but not in both. The set Q contains all linear combinations of cycles. Merged with cycles of the basis, these linear combinations represent a way to merge more than two cycles of the basis.

At each iteration of Algorithm 5.2, new simple cycles are in the set $R \cup \{B\}$. To avoid waiting for the enumeration of all cycles, an operation performing redundancy check is added on line 18. For each new cycle, if the cycle contains the meta-edge to analyze, the

Algorithm 5.2 Enumeration of combinations of meta-edges

```
1  procedure Enumerate-Combinations(Dual 𝔐*, v ∈ V_𝔐*)
2     {B_1, ..., B_n} ← Paton-Algorithm(𝔐*)
3     Q ← {B_1}, R ← ∅, R* ← ∅
4
5     for B ∈ {B_2, ..., B_n} do
6        for T ∈ Q do
7           if T ∩ B ≠ ∅ then
8              R ← R ∪ {T ⊕ B}
9           else
10             R* ← R* ∪ {T ⊕ B}
11          end if
12       end for
13       for U, V ∈ R do          // Remove non-simple cycles from R
14          if U ⊂ V then
15             R ← R \ {V}, R* ← R* ∪ {V}
16          end if
17       end for
18       for C ∈ R ∪ {B} do       // New valid cycles are in R ∪ {B}
19          if v ∈ C and Replace-MetaEdge(v, C) then
20             remove v and its incident edges from 𝔐*
21             break
22          end if
23       end for
24       Q ← Q ∪ R ∪ R* ∪ {B}
25       R ← ∅, R* ← ∅
26    end for
27    return S
28 end procedure
```

check can be performed. If a cycle makes the meta-edge redundant, the algorithm can be exited earlier and the meta-edge is removed.

5.4.3.3. Use of Cycle Enumeration in the Procedure

Each cycle can be written as a finite sequence $(m_1, \ldots, m_n)$ of meta-edges such that $m_1 = m_n$ is the meta-edge to be analyzed. This cycle contains a candidate combination to check if m_1 is redundant. In other words, it is now possible to test whether the combination of meta-edges $(m_2, \ldots, m_{n-1})$ can replace m_1 in terms of consistency preservation.

5.4.4. Detection of Redundant Rules

Given a meta-edge to analyze and an alternative combination of meta-edges, the last step of the decomposition procedure is to determine if the meta-edge can be replaced by the combination without altering consistency. This is the core of the decomposition procedure, i.e. the algorithm by which decomposition is performed or not.

As stated during the construction of the metagraph in Section 5.3.4.1, this decision can be made by processing QVT-R domain patterns as equations represented by formulae of first-order logic. Programmatically, this requires the use of an *automated theorem prover*. The fact that a meta-edge is redundant, i.e. that it can be removed from the metagraph, depends on the output of the prover embedded in the decomposition procedure.

5.4.4.1. Combinations as Horn Clauses

Let $(m_1, \ldots, m_n)$ be a candidate combination of meta-edges to replace a meta-edge m. The set of consistency rules associated with a meta-edge m is denoted $c(m)$. For m to be redundant, instances of metamodel elements fulfilling the combination $(c(m_1), \ldots, c(m_n))$ must also fulfill the set $c(m)$ of consistency rules. In other words, if an instance fulfills the combination, it necessarily fulfills the meta-edge m. Hence, m is redundant.

This implication can be written as a first-order formula, or more precisely as a Horn clause. In terms of logic, the implication must be valid in order to remove m without altering the consistency specification.

$$(c(m_1) \land c(m_2) \land \cdots \land c(m_n)) \implies c(m)$$

In the Horn clause above, consistency rules on the left-hand side, i.e. those of the combination, are called *facts*. Consistency rules on the right-hand side, i.e. those of the meta-edge to analyze, form the *goal*. Intuitively, this implication represents the deduction of $c(m)$ from $c(m_1), \ldots, c(m_n)$. Proving that such a formula is valid is however difficult. This means that the formula has to evaluate to TRUE, whatever the values of metamodel elements and QVT-R variables in consistency rules.

A common workaround uses the following property: proving that a Horn clause H is valid is actually equivalent to proving that its negation $\neg H$ is unsatisfiable. Conversely, $\neg H$ being satisfiable means that H is not valid. That is, the goal cannot always be deduced from the facts. Regarding consistency preservation, this means that there are instances

fulfilling the combination $(c(m_1), \ldots, c(m_n))$ that do not fulfill $c(m)$. Consequently, the combination is not sufficient to replace m, which cannot be removed from the metagraph.

Using the negation of the Horn clause for the resolution is called *refutation*. It amounts to a proof by contradiction. This is common technique in theorem provers, thus referred to as *refutational*. The contradiction comes from the structure of the resulting formula.

$$\begin{aligned} \neg H &\equiv \neg \left[(c(m_1) \wedge c(m_2) \wedge \cdots \wedge c(m_n)) \implies c(m) \right] \\ &\equiv \neg \left[\neg (c(m_1) \wedge c(m_2) \wedge \cdots \wedge c(m_n)) \vee c(m) \right] \\ &\equiv c(m_1) \wedge c(m_2) \wedge \cdots \wedge c(m_n) \wedge \neg c(m) \end{aligned} \tag{5.1}$$

The formula above is satisfiable if both facts and the negation of the goal are satisfiable, hence the proof by contradiction. In terms of automated theorem proving, each operand in the logical conjunction is called an *assertion*.

In the decomposition procedure, each set of consistency rules from the meta-edge to analyze and meta-edges in the combination is translated into a conjunction of first-order formulae (see Chapter 6 for details on translation). This conjunction models a system of equations, as depicted in Section 5.3.4.2. Then, according to Formula 5.1, each formula representing a set of rules is added to the stack of assertions of the prover, except the one of the meta-edge to analyze which is first negated and then added to the stack.

There is an important point in the implementation of Horn clauses for the decomposition procedure. Horn clauses are generally described without quantifiers. That is, translated formulae in consistency rules can be first-order formulae but there is no explicit quantification of all metamodel elements and QVT-R variables in the clause.

In fact, the quantification is implicit and can be defined as follows. Unless explicitely quantified, all variables are universally quantified ($\forall$). Translation of consistency rules generate two types from (first-order) variables: metamodel elements and QVT-R variables. Metamodel elements being shared among meta-edges, they appear both in facts and the goal. However, QVT-R variables local to the analyzed meta-edge first (and only) appear in the goal.

Universally quantified QVT-R variables in the goal have the following meaning. For the whole Horn clause to be valid, all QVT-R values must match the values of metamodel elements. Clearly, this is not the way pattern matching operates in QVT-R. As stated in Section 5.3.4.2, consistency of the rule depends upon the existence of a solution, i.e. an assignment of QVT-R variables so that all patterns are fulfilled. Consequently, it is necessary to add an explicit existential quantifier ($\exists$) for QVT-R variables in the goal.

Example 5.4.1 *Let e_1, e_2 and e_3 be attributes of metaclasses from three metamodels. We assume that consistency is preserved if the three attributes have the same value in an instance. Consequently, there are three consistency rules: $e_1 = e_2$, $e_2 = e_3$ and $e_1 = e_3$.*

Suppose that we want to show that $e_1 = e_3$ is redundant through the combination of $e_2 = e_3$ and $e_1 = e_3$. Here are three proposals for Horn clauses:

(1) Without QVT-R variable: $(e_1 = e_2) \wedge (e_2 = e_3) \implies (e_1 = e_3)$

(2) With a QVT-R variable v: $(e_1 = e_2) \land (e_2 = e_3) \implies ((e_1 = v) \land (e_3 = v))$
(3) With v quantified: $(e_1 = e_2) \land (e_2 = e_3) \implies (\exists\, v : (e_1 = v) \land (e_3 = v))$

Clause (1) is valid because equality is a transitive relation. Using QVT-R, metamodel elements are bound to QVT-R variables so $e_1 = e_3$ is expressed (and translated) as $(e_1 = v) \land (e_3 = v)$. This leads to Clause (2) which is no more valid. For example, the clause evaluates to FALSE *with the interpretation $\{e_1 = e_2 = e_3 = 0, v = -1\}$. As a solution, Clause (3) encodes the fact that one assignment of QVT-R variables is enough. Therefore, free variables in the goal are existentially quantified and Clause (3) is valid.*

In conjunction with the use of quantifiers, another solution to the problem of free QVT-R variables consists in exploiting the transitivity of relations to remove QVT-R variables. For example, computing the transitive closure in the Clause (2) of Example 5.4.1 can lead to Clause (1). The major drawback is that not all OCL operations are transitive. Therefore, not all QVT-R variables can be processed this way, unlike the use of a quantifier.

5.4.4.2. Additional Conditions

In addition to property template items, consistency can be specified with QVT-R using conditions, i.e. preconditions (`where` clauses), invariants (`when` clauses) and local invariants (clauses in domain patterns). These constructs can be translated as well. Given the restriction of Section 5.3.4.3, we assume that conditions are only made up of variables.

First, invariants and local invariants are conditions that must be fulfilled by metamodel elements at any time, just as property template items. Therefore, they can be embedded in Horn clauses along with consistency rules. The approach to support invariants is as follows. If a consistency rule uses a QVT-R variable that is also part of an invariant, then the invariant must be added as a conjunction next to the rule. If the rule is a fact (resp. a goal) of the Horn clause, the invariant is added as a fact (resp. a goal) of the clause.

Intuitively, invariants model additional constraints on QVT-R variables. If a QVT-R variable appears in the Horn clause, all related constraints must also appear in the clause.

Then, preconditions must be processed differently. During the execution of a QVT-R transformation, a precondition that is not fufilled prevents the associated QVT-R relation from being executed. In this work, preconditions are restricted and only consist of QVT-R variables. Consequently, either preconditions are satisfiable, in which case rules and invariants of the same QVT-R relation can be processed or they are unsatisfiable. If they are unsatisfiable, the QVT-R relation must not be processed, i.e. associated rules and invariants must not appear in the metagraph.

Therefore, for each QVT-R relation with a `when` clause, its precondition can be translated during the construction of the metagraph and verified with the automated theorem prover. If the precondition is satisfiable, the relation is processed. Otherwise, the relation is ignored.

Regarding the existential quantification of QVT-R variables, additional conditions behave in the same way as the usual property template items. All QVT-R variables that first appear in the right-hand side of the Horn clause must be existentially quantified.

5.4.4.3. Interaction with the Automated Theorem Prover

Given a set of assertions representing consistency rules, the automated theorem prover can provide three answers: `SAT`, `UNSAT` or `UNKNOWN`.

- `SAT` means that the prover found an assignment in Formula 5.1 such that the formula is satisfiable. This indicates that the meta-edge *m* cannot be entirely deduced from the alternative combination of meta-edges. As a result, *m* is not redundant and cannot be removed from the graph.
- `UNSAT` means that Formula 5.1 is unsatisfiable, i.e. there is a proof by contradiction that the meta-edge *m* can be deduced from the combination. This makes *m* redundant. The meta-edge and can be removed from the metagraph.
- `UNKNOWN` means that the prover was unable to find an assignment to Formula 5.1 as well as to prove that it is unsatisfiable. There are two main reasons for this: either the formula lies in a non-decidable fragment of first-order logic or the prover had not enough time to perform the verification. In both cases, conservativeness requires not to remove the meta-edge *m* from the metagraph because it is not absolutely certain that it could have been redundant.

If the automated theorem prover answers `SAT`, the analyzed meta-edge is removed from the metagraph. The decomposition procedure stops when each meta-edge has been tested once. Note that if the connected component becomes a tree after a few removals of meta-edges, then the last tests of remaining meta-edges are trivial. As there are no more cycles in the dual of the connected component, the automated theorem prover is not used.

The result of the decomposition procedure is a set of connected components of the metagraph, each connected component being also associated with a set of removed meta-edges. This structure (made of remaining edges on one hand and of removed edges on the other hand) corresponds to a decomposition. The use of the metagraph as described in previous subsections is finally illustrated by means of an example.

Example 5.4.2 *Using the dual of the metagraph of Figure 5.9, this example shows the translation of meta-edges to generate a Horn clause. Figure 5.12 shows a cycle where the meta-edge to analyze is the meta-edge 3. The candidate combination to replace it is (1, 2).*

In the resulting Horn clause, meta-edges 1 and 2 are the facts whereas meta-edge 3 is the goal. Property template items associated with meta-edges lead to the following formula:

$$
\begin{aligned}
&[(\texttt{Person::firstname} = \mathit{f1}) \wedge (\texttt{Person::lastname} = \mathit{l1}) \wedge\ (\texttt{Resident::name} = \mathit{f1}\texttt{+' '+}\mathit{l1})] \\
\wedge\ &[(\texttt{Person::firstname} = \mathit{f2}) \wedge (\texttt{Person::lastname} = \mathit{l2}) \wedge\ (\texttt{Employee::name} = \mathit{f2}\texttt{+' '+}\mathit{l2})] \\
\Longrightarrow\ &(\exists\, n : (\texttt{Resident::name} = n) \wedge (\texttt{Employee::name} = n))
\end{aligned}
$$

QVT-R variables (`fstn`, `lstn`) have been renamed to avoid conflicts. This is necessary because they are no longer isolated as they were before in two distinct QVT-R relations.

The formula above is valid. Therefore, the meta-edge 3 is redundant and can be removed from the dual. The resulting graph is made up of two vertices and one edge. As a result, it is a tree so the decomposition for this connected component is optimal.

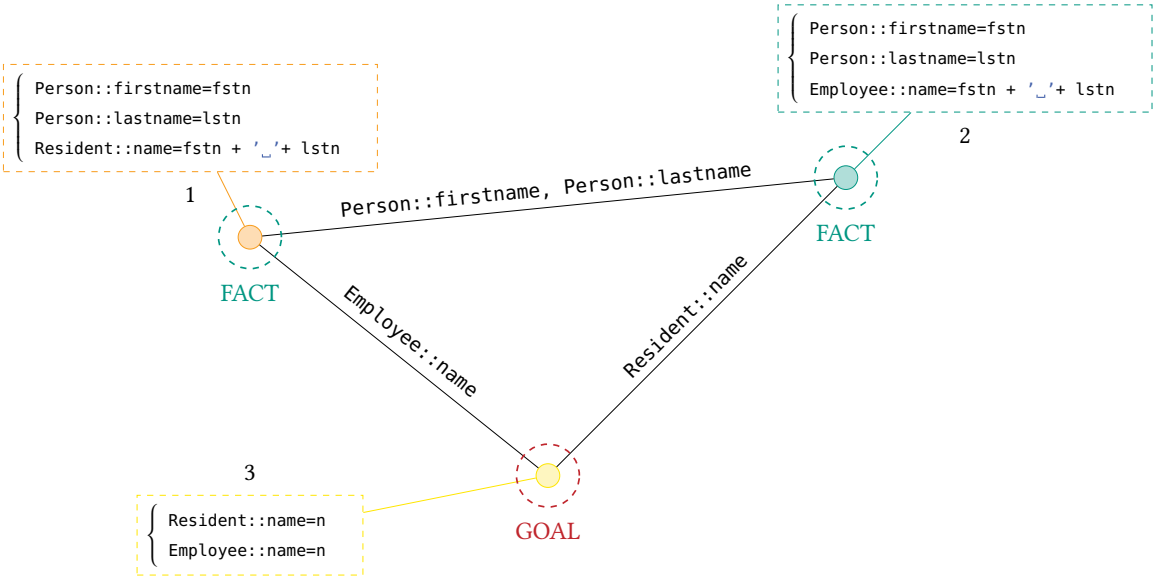


Figure 5.12.: Candidate combination $(1, 2)$ in a dual

6. Constraint Translation

An important part of this thesis lies in the use of an automated theorem prover to identify sources of redundancy in consistency specifications. The transition from consistency rules to logic formulae, called *constraint translation*, represents a paradigm change whose implementation is presented here.

The first section explains the reasons for the choice of automated theorem proving over other methods of symbolic computation to reason about OCL expressions and QVT-R constructs in the context of the decomposition procedure. It also mentions the limitations of such a choice. The second section describes the basic correspondence between types in OCL and types in Z3, the chosen automated theorem prover. The third section presents a way to implement some OCL collections, i.e. data structures, in Z3. Finally, the fourth section focuses on the translation of OCL operations.

6.1. Symbolic Computation for OCL and QVT-R

6.1.1. Automation of the Decomposition Procedure

The need for a tool for symbolic computation to implement decomposition methods has been described in Section 5.3.4.1. There exist many approaches to perform symbolic computation with the aim of analyzing programs, e.g. in the context of *formal methods* for the specification and the verification of software systems.

An important factor for the classification of tools for symbolic computation is *automation*. Depending on the desired level of automation of the decomposition procedure, some tools are more suitable than others.

- *Automated decomposition procedure.* The procedure is called *automated* if the optimization of consistency specifications can be performed without the intervention of humans. That is, users do not make any decision on how to perform the decomposition. In this case, tools for symbolic computation are automated too.

- *Semi-automated decomposition procedure.* The procedure is called *semi-automated* if it interacts with the user during the decomposition. Most often, the user makes decisions that the procedure cannot make alone, e.g. checks for which there is no algorithm. *Interactive* is sometimes used as a synonym for semi-automated.

These approaches have both advantages and drawbacks. This is often a compromise between automation and capability. The less automated a procedure is, the more likely it is to involve a user and thus solve difficult cases. The approach adopted in this thesis is to

use an automated tool, so that the decomposition procedure is also automated. The main developments and consequences of a semi-automated decomposition procedure compared to the one of this thesis are discussed in Section 9.2.

6.1.2. Choosing an Approach for Constraint Translation

The selection of a tool for symbolic computation now has to be made among automated tools. Approaches taken into account were *automated theorem proving* [Lov16], *model finding* [TJ07], *answer set programming* [Lif08] and *constraint logic programming* [JL87].

Due to the nature of OCL expressions, there are some requirements that the chosen tool should meet. In particular, deciding factors for the decomposition procedure are:

- The support of primitive datatypes, operations and data structures of OCL. For example, some tools in formal methods are said to be *high-level*, meaning that they have an high level of abstraction and that their manipulation by end-users is easier;
- The complexity of the translation. This can be characterized by the effort required to map OCL and QVT-R to the tool and to give an interpretation to its results.
- The ability to check the validity of formulae, i.e. not only the production of refutations. This is required to show that a consistency relation is redundant. In such a case, the property must hold for *all* instances of the consistency relation, hence the need of a prover.

The following subsections provide an overview of the position of the above-mentioned approaches with respect to these factors.

6.1.2.1. OCL and Automated Theorem Proving

Theorem provers are tools that take well-formed (logical) formulae as inputs and try to determine if formulae are satisfiable. There are two categories of automated theorem provers. First, *general-purpose theorem provers* (GPTP) take first-order formulae (usually, a set of axioms and a conjecture) and search for a proof. They are low-level tools regarding the needs of the decomposition procedure as they do not provide a native support for arithmetic, data structures, etc.

Second, *theory-specific theorem provers* (TSTP) take first-order formulae with respect to some theory. For this reason, they are also known as satisfiability modulo theories (SMT) solvers. The fact that binary variables can be replaced with predicates of the theory makes this approach interesting for the decomposition procedure. For example, some theories are able to give results on string manipulation or arithmetic. This provides an easier mapping to OCL. A drawback of SMT provers is that it is hard to define extra theories. This means that the scope of the decomposition procedure would then be greatly limited to what the SMT natively supports.

6.1.2.2. OCL and Model Finding

Model finding is a frequent approach in software engineering. For example, model finders are used in model repair and model transformation. They are able to interpret Ecore and UML metamodels as well as transformation languages such as QVT-R or ATL. They behave more like "refuters" than "provers", in the sense that they search for counterexamples rather than searching for proofs of validity. This makes model finders incompatible with the decomposition procedure.

6.1.2.3. OCL and Answer Set Programming

Being a subset of logic programming, *answer set programming* (ASP) works with facts and deductions. The user declares facts and waits for answer sets, i.e. model solutions. In the context of the decomposition procedure, the consistency rules of an alternative combination of consistency relations are facts and the examined consistency rule can be regarded as redundant if it can be deduced from facts. Just like GPTP and model finders, ASP does not provide a built-in support for primitive data types of OCL. This implies that all axioms of arithmetic, string manipulation, operations, etc. should be declared as facts, which is limiting for the decomposition procedure.

6.1.2.4. OCL and Constraint Logic Programming

Finally, *constraint logic programming* (CLP) is another subset of logic programming. It can be regarded as a merge of constraint satisfaction – the field of application of constraint networks – and logic programming. Since facts in CLP can be organized as constraints, this represents an interesting gain in expressiveness for the decomposition procedure. Moreover, it is possible to extend the set of supported constraints. However, some aspects of constraint solvers are disadvantages for the procedure. All checks are made on a finite set of value and solvers strongly recommend to define an initial domain in which the expected values lie. Such a domain cannot be easily inferred from OCL constraints.

In the light of the above commments and the important deciding factors, an approach using SMT solvers for the decomposition procedure was chosen. It provides a rather natural correspondence with primitive datatypes in OCL, a high-level formalism to encode rules and it is able to prove the unsatisfiability of certain formulae.

Note that there is no perfect solution. For example, CLP solvers would be better suited for consistency specifications representing discrete optimization problems (which should rarely happen in practice). As a result, an ideal choice would combine several approaches, including interactive approaches.

The SMT solver used in the implementation of the decomposition procedure is Z3[1], a theorem prover whose input format is defined by the SMT-LIB standard[2].

[1] `https://z3prover.github.io`
[2] `http://smtlib.cs.uiowa.edu/`

6.1.3. Theorem Proving for Decomposition

Regardless of the tool chosen, there are theoretical and practical limitations to OCL such that not all OCL expressions can be used to decide whether a consistency relation is redundant. Theoretically first, it was shown by Beckert et al. that OCL can be translated into first-order predicate logic [BKS02]. First-order logic is undecidable, i.e. there is no procedure that verifies if a first-order formula is valid. Moreover, OCL formulae being essentially full first-order formulae [BCD05], they do not form a decidable fragment of the logic. As a result, OCL is undecidable in general.

Practically then, it is easy to find an OCL constraint whose translation into a first-order formula cannot be proved valid using Z3. For example, constraints making use of data structures are often translated into formulae with quantifiers. In current SMT solvers, the support of formulae with quantifiers is limited. Section 6.4.8 gives some examples of OCL operations that cannot be easily implemented or decided by Z3.

On the plus side, however, theories in Z3 often ensure that some subsets of first-order logic are decidable. After the translation, many OCL constraints result in formulae that are part of these subsets. For example, reasoning about strings in Z3 leads to good results in practice. Consequently, reasoning on QVT-R transformations through SMT solvers is a case-by-case matter. Theoretical limitations are not a definitive obstacle to their use.

6.2. Primitive Datatypes

Table 6.1 shows how primitive datatypes can be translated from OCL to Z3. It also shows how datatypes in Ecore metamodels are mapped to datatypes in OCL constraints. In Z3, types are called *sorts*. Every expression in Z3 has a sort. Moreover, the sort of an operation can be determined by the sort of its operands.

OCL has an *UnlimitedNatural* datatype to represent multiplicities. The difference between *UnlimitedNatural* and *Integer* is that the former can have an infinite value whereas the latter cannot. This subtelty was not translated in the context of the decomposition procedure. This could be done with a composite sort, e.g. a couple (IntSort, BoolSort) where the value is ∞ if the boolean is TRUE, or equal to the integer otherwise. Introducing such a sort can however have consequences for the decidability of formulae in Z3.

OCL datatype	**Ecore datatype**	**Z3 datatype**
Integer	`EInt`	IntSort
Real	`EDouble`	RealSort
Boolean	`EBoolean`	BoolSort
String	`Estring`	StringSort
UnlimitedNatural	`EInt`	IntSort *(without infinity)*

Table 6.1.: Correspondence between primitive datatypes in OCL, Ecore and Z3

6.3. Data Structures

6.3.1. Collection Literals

Primitive data structures in OCL are called *collections*. The term *collection literals* refers to OCL expressions that represent data structures with constant and pre-defined values, e.g. `Sequence{1, 4, 9}` or `Set{2, 5}`. There are four types of collections depending on two features. First, whether elements are ordered in the collection (Ordered). Second, whether duplicate elements are allowed in the collection (Unique).

As a result, OCL provides `Sequence(T)` (Ordered, not Unique), `Set(T)` (not Ordered, Unique), `Bag(T)` (not Ordered, not Unique) and `OrderedSet(T)` (Ordered, Unique). Among those four collections, a support for collection literals of `Sequence(T)` and `Set(T)` was provided in the implementation of the decomposition procedure.

It is noteworthy that collection literals rarely occur in consistency specifications. In general, collections are groups of objects represented by role names (as a consequence of the existence of references in metaclasses).

6.3.1.1. Sequence Literals in Z3

In SMT solving, the fundamental theory to represent several values at the same time is the theory of *arrays*. The idea is to represent an array as a map whose domain is a set of indexes and whose codomain is a set of values. For example, the sort `(Array Int Real)` represents an array whose indexes are integers and values are real numbers. Everything is function in Z3. As a consequence, arrays are purely functional data structures that are immutable.

The theory of arrays also define two functions called `select` (to read the value stored at a given index) and `store` (to write a given value at a given index). The strategy to build a sequence literal (e.g. `Sequence{1, 4, 9}`) is to recursively call `store` operations for each element of the literal. For example, the literal `Sequence{1, 4, 9}` can be translated into the Z3 expression of Listing 6.1 where `(store arr i v)` writes the value `v` at index `i` in the array `arr`.

```
1  (declare-const arr (Array Int Int))
2  (store (store (store arr 0 1) 1 4) 2 9)
```

Listing 6.1: Recursive construction of an OCL sequence using a Z3 array

The problem of the array of line 2 of Listing 6.1 is that there is no indication of the sequence length. Such a value is not available natively with Z3 because arrays are represented as maps. For each element of the domain of the array, there is a corresponding value in the codomain. As a result, the naive length of the array is the cardinality of the domain.

This information must be encoded during the translation. The most convenient way to represent data structures in Z3 is to use *algebraic datatypes*. An algebraic datatype is a composite type, i.e. a type built by combining Z3 primitive datatypes. To represent an OCL

sequence, we combine an `Integer` to store the length of the sequence and an `Array Int T` to store values of the sequence, where `T` is the type of objects in the sequence.

The datatype sort is composed of a unique name and a constructor. The fact that collections in OCL are parametric, i.e. that a `Collection(T)` can only contain elements of type `T`, is simulated by adding the parameter in the name of the datatype sort. For example, a sequence of integers in OCL is translated as a `Sequence<Integer>` sort.

Then, the constructor consists of two things. First, a *constructor name*, i.e. the name of the function invoked to use the new datatype sort. This name is `mkSequence` for all datatype sorts representing OCL sequences. Then, *field names*, i.e. names of functions to get the content of fields. For a sequence, these fields are `length` and `array`.

```
1  (declare-const arr (Array Int Int))
2  (declare-datatypes () ((Sequence<Integer> (mkSequence
3      (length Int)
4      (array (Array Int Int))))))
5
6  (mkSequence
7      3 ; length (computed during the translation)
8      (store (store (store arr 0 1) 1 4) 2 9) ; array
9  )
```

Listing 6.2: Recursive construction of an OCL sequence using a Z3 algebraic datatype

6.3.1.2. Set Literals in Z3

The support of sets in Z3 is also based on arrays. An OCL set that contains objects of type `T` is represented as a Z3 array that maps values of type `T` to booleans. If an element e is in the set, then the boolean at the index e is TRUE.

As a result, the representation of set literals is very similar to that of sequence literals. The unique name for the datatype sort is now `Set<T>` and the constructor name is `mkSet`. Fields stay the same: `length` contains the size of the set and `array` the values in the set. The Z3 reference also explains how main set operations (intersection, union, difference, etc.) can be implemented when sets are represented as arrays [MB11].

6.3.2. Collections from Role Names

As stated in Section 6.3.1, collection literals are, however, rarely used in consistency specifications. The reason for this is that the role of consistency specifications is to relate metamodel elements. Literals, either primitive literals or collection literals, are not necessary to bind elements from different metamodels.

Another use of collections is implicit. In domain patterns, the left part of a property template item can represent an attribute or a role name, i.e. an alias for objects at the end of the reference owned by the metaclass of the pattern. If the upper bound of the multiplicity of the reference is bigger than one, e.g. `1..5` or `0..*`, then the role name may

represent a collection of objects. The returned collection type depends on whether the end is ordered and/or unique (see Section 6.3.1).

When a static analysis is performed on property template items, the left part is first analyzed. If it is a role name, it is also embedded as a `EReference` in the Ecore metamodel. As a result, the following information is associated with the role name: the type (`eType`) of objects behind the role name, the lower and upper bounds of the multiplicity and boolean properties indicating whether the reference is ordered and unique. As long as the QVT-R relation is not executed, there is no way to list objects of the role name. Thus, the translation of collection literals presented in Section 6.3.1 is useless.

Nevertheless, it is still possible to reason about role names by means of *uninterpreted functions* (UF) in Z3. Role name r of an metaclass c can be represented as a function of c, e.g. $r(c)$. This function is uninterpreted in the sense that r is only a symbol and has no meaning. However, it is sometimes sufficient. For example, let c_1 and c_2 be two QVT-R variables of the same metaclass. This metaclass has a reference r, represented in Z3 as an uninterpreted function. If a consistency rule ensures that $c_1 = c_2$, then it follows that $r(c_1) = r(c_2)$. There are efficient decision procedures for UF in current SMT solvers. Complex formulae are often simplified in the first place with UF instead of symbols. For example, string concatenation in OCL can be replaced with an UF `concat`. The manipulation of strings is therefore replaced by that of symbols. The reason for this is that if the approximated formula is unsatisfiable, the original formula is unsatisfiable too [KS].

6.4. Operations

EssentialOCL – the subset of OCL that QVT-R embeds – provides many primitive operations, either for primitives datatypes such as **`Integer`**, **`String`**, etc. or for collections such as **`Set`**`(T)`, **`Sequence`**`(T)`, etc.

This section presents the translation from OCL to Z3 of operations on primitive datatypes or data structures. Not all OCL operations have been implemented in this thesis. There are two reasons for this. First, some operations are secondary to the implementation of a prototype of the decomposition procedure. Second, not all operation can be translated by means of Z3. In the latter case, the last section explains the main reasons for this.

In OCL, an operation has a source and zero or more arguments. Even among primitive operations, the source is important to distinguish operations that have the same name. For example, the `+` operation denotes an addition when the source is an **`Integer`** but concatenation when the source is a **`String`**.

6.4.1. Arithmetic Operations

See Appendix A.1 for the translation of arithmetic operations.

Translated arithmetic operations are the addition (`+`), the subtraction (`-`), the multiplication ($*$), the euclidean division (`div`), the division (`/`), the modulo (`mod`) and the absolute value (`abs`). The support of arithmetic in Z3 is based on integers (IntSort) and reals (RealSort). Contrary to OCL, reals are not automatically converted into integers, there is a Z3 function for this.

6.4.2. Boolean Operations

See Appendix A.2 for the translation of boolean operations.

Translated boolean operations are the negation (`not`), the conjunction (`and`), the disjunction (`or`), the exclusive disjunction (`xor`) and the implication (`implies`). The support of boolean operations in Z3 is based on BoolSort, the boolean type of Z3. Contrary to OCL, the `invalid` value does not exist in Z3. Therefore, invalid values are not translated in the truth tables of Z3 boolean functions.

6.4.3. Conversion Operations

See Appendix A.3 for the translation of conversion operations.

Translated conversion operations are the floor function (`floor`) and the round function (`round`). The conversion of real to integers in Z3 is not automatic. Z3 provides a function `to_int`, defined by the SMT-LIB as a function that maps a real number to its integer part. This function is used in the definition of `floor` and `round`.

6.4.4. Equality Operators

See Appendix A.4 for the translation of equality operators.

Translated equality operators are the equal-to operator (=) and the not-equal-to operator (<>). In Z3, the equality operator (=) also replaces the double implication operator. It is part of the core theory of Z3 but its exact behaviour is theory-specific and depends on the argument types. For example, equality between two arrays in Z3 is encoded as an axiom called *extensionality*.

6.4.5. Order Relations and Extrema

See Appendix A.5 for the translation of order relations and extremum functions.

Translated order relations and extremum functions are <, <=, >, >=, the minimum function (`min`) and the maximum function (`max`). The lack of built-in functions for `min` and `max` is solved by the Z3 `if-then-else` function.

6.4.6. Collections Operations

See Appendix A.6 for the translation of operations related to collections.

Translated operations related to collections are the is-empty function (`isEmpty`), the is-not-empty function (`notEmpty`), the object inclusion (`includes`) and the object exclusion (`excludes`). Specific to sequences are there the get-first function (`first`), the get-last function (`last`) and the get-nth function (`at`). Specific to sets are there the union, the intersection, the difference (`excludesAll`) and the symmetric difference.

No verification is performed on the length of sequences to retrieve elements.

6.4.7. String Operations

See Appendix A.7 for the translation of operations related to strings.

Translated operations related to strings are the concatenation (`concat` or `+`), the substring extraction (`substring`), the length (`size`) and the conversion to integer (`toInteger`).

6.4.8. Untranslatable Operations

Operations of the OCL reference are said to be *untranslatable* if state-of-the-art SMT solvers (such as Z3) do not provide a way to reason about them. There are multiple reasons behind the existence of untranslatable operations. We survey an example of such an operation and provide ideas to overcome this limitation.

The OCL reference includes two primitive operations for the `String` data type that convert characters of a string to upper case (`toUpper`) or to lower case (`toLower`). Although simple, these operations cannot be easily translated with Z3.

The reason for this is that strings are represented as sequences in Z3. In automata theory, an automaton that takes a sequence of symbols and returns another sequence of symbols is called a *finite-state transducer*. It can be regarded as an extension of usual automata that only returns whether a word is accepted or not. There already exist decision procedures for finite-state transducers [Vea+12]. They are however not integrated in Z3 yet, one reason being that Z3 reasons about sequence constraints in a different way.

The best solution so far would be to use an alternative tool, built for reasoning about string encoding and decoding. We discuss in Section 9.2.2 the integration of multiple symbolic computation tools into the decomposition procedure.

This example illustrates that even simple operations in high-level languages may require important changes in SMT solvers.

7. Evaluation

The decomposition procedure presented in this thesis resulted in a prototype. This prototype makes it possible to test the practical aspects of the decomposition of consistency relations. In this chapter, we aim to evaluate the decomposition procedure as a whole.

Two aspects of this approach will be evaluated: the *functional correctness* of the procedure and its *applicability*. First, we describe the methodology of our evaluation. This includes a strategy to address the research questions stated in Section 1.2, as well as a description of the evaluation material, i.e. the data used to conduct the evaluation.

Then, each aspect of evaluation is covered in a section. The functional correctness is evaluated through the analysis of the decomposition procedure. Here, we assess whether the procedure meets some functional requirements, i.e. the generation of a tree-based consistency specification when such a specification exists. The applicability is evaluated through the use of the prototype. The aim is to find out if the procedure can be used outside of this thesis and applied to consistency specifications in other contexts.

The last section discusses the results obtained. From the evaluation, the benefits and limitations of the decomposition procedure with respect to the initial research question are summarized. Finally, ideas for further evaluation are proposed.

7.1. Methodology

In this section, we present the methodological approach to this evaluation. We first recall the research questions of this thesis and introduce a way to answer them. Then, we detail elements that can be used in order to perform the evaluation.

7.1.1. Addressing Research Questions

Research questions raised in Section 1.2 focus on three perspectives: the definition of decomposition for multi-model consistency preservation (**Q1**), the suitability of QVT-R to express and decompose consistency relations (**Q2**) and the design of a decomposition procedure (**Q3**). Addressing these three questions should make it possible to meet the research goal of this thesis; that is, the identification of decomposable relations by means of abstract decomposition methods and their implementation using the QVT-R transformation language (**G1**).

An evaluation to determine whether the goal was reached can be conducted according to two approaches that complement one another.

- *Theoretical evaluation.* The evaluation uses mostly qualitative arguments. It is performed by analyzing concepts and definitions used throughout this thesis;

- *Empirical evaluation.* The evaluation uses mostly quantitative arguments. It is performed by collecting test results, interpreting them and defining a metric to measure the results achieved against the expected results.

Both approaches are combined in this evaluation. First, we assess the *functional correctness* of our approach using tools for a theoretical evaluation. Therefore, evaluation is performed by construction, i.e. by reasoning about the design of decomposition methods and the decomposition procedure.

Then, we assess the *applicability* of our approach using tools and collected data for an empirical evaluation. Evaluation is performed by running the implementation of the decomposition procedure with various example scenarios. These scenarios must generally reflect what the procedure should be able to deal with.

7.1.2. Evaluation Material

According to the previous approaches, we gather the evaluation material from two sources.

- *Description of the decomposition procedure.* This includes the functioning the procedure, the algorithms used by the procedure (along with their time complexity), formal properties required for the implementation of the procedure, the code of the prototype and standards of languages (such as QVT-R and EssentialOCL);
- *Execution of the decomposition procedure.* This includes example scenarios (i.e. example specifications written with QVT-R), execution results and error handling.

The description of the decomposition procedure is mainly used to gauge the functional correctness of the procedure, whereas example scenarios and their execution are mainly used to survey the applicability of the decomposition procedure.

7.2. Functional Correctness

The functional correctness of an algorithm is a property asserting that the input-output behaviour of the algorithm corresponds to the specification. In the context of the decomposition procedure, the functional correctness refers to the fact that output specifications represent valid decompositions of input specifications.

To this end, we defined two formal properties that the decomposition procedure must fulfill in Section 4.3. First, *conservativeness* requires for a given set of metamodels that the set of consistent instances be the same for the input and the output specifications. In other words, the decomposition procedure does not alter the specification. Second, *usefulness* requires the existence of a decomposition in the output algorithm, i.e. the output specification is in a way more applicable than the input specification.

Satisfying both formal properties is a minimal requirement. However, the decomposition procedure should ideally be able to return the tree (i.e. the optimal) decomposition if such

a decomposition exists. To evaluate the functional correctness of the decomposition procedure, we proceed as follows. First, the ability of the procedure to return the optimal decomposition when such a decomposition exists is evaluated. Then, we check that in all cases, the result provided by the decomposition procedure satisfies both formal properties. In this way, it can be checked that the procedure cannot degrade the specification.

7.2.1. Finding Existing Tree-Like Specifications

A tree specification is a specification whose consistency relation graph is made up of independent trees. More generally, a tree-like specification is a specification whose consistency relation graph is made up of independent subgraphs that resemble trees. Here, "resemble" means that the subgraph is sparse. For the reasons stated in Section 4.1.2, tree-like specifications are preferred in the context of consistency preservation. Furthermore, a tree specification is always regarded as optimal following the same reasons. As a consequence, concepts related to decomposition in this thesis were defined in order to foster the appearance of independent trees.

7.2.1.1. Trees and Decomposition Methods

We first introduced three decomposition methods in Section 4.2. Given the consistency relation graph of a specification, these methods modify the graph in two ways. Either they remove edges from the graph or they separate a graph into independent subgraphs. We now evaluate whether these methods produce optimal decompositions.

The first method separates independent subgraphs. It does not make resulting subgraphs more sparse than the initial graph, i.e. subgraphs cannot be transformed into trees using this method. However, it ensures that each subgraph is connected, which is a prerequisite to obtain trees. We asserted that this method was applied once, by computing connected components of the consistency relation graph. After the application of the method, there cannot exist a disconnected subgraph. There are therefore no osbtacles to find an optimal decomposition in the application of the first decomposition method.

The second method removes consistency relations that are said to be totally redundant. For a relation to be removed, there must exist an alternative path between the two endpoints of the relation. As a result, the subgraph in which the relation is removed always remains connected. The gradual removal of edges in a graph under the condition that it remains connected is a known algorithm for finding the spanning tree of a graph [Kru56]. It is known as the *reverse-delete* algorithm.

The third method is used at the same time as the second. It splits a subgraph into two independent subgraphs according to consistency relations that are said to be partially redundant. In this case, the consistency relation is divided into two parts. The first part of the consistency relation is the one that is contained in at least one alternative combination of relations. The second part of the consistency relation is the independent part, i.e. the part that is not contained in any alternative combination. In terms of graphs, the two parts lead to two independent subgraphs. The first independent subgraph is equal to the original subgraph from which the first part of the consistency relation was removed as it can be replaced with alternative combinations of relations. As a result, this subgraph is

more sparse than the original subgraph. The second independent subgraph is by definition a tree since it only composed of one independent consistency relation, the second part of the initial consistency relation.

Given a connected subgraph of consistency relations, the simultaneous application of the second and the third methods ensures that the optimal decomposition is returned. The reason for this is that each consistency relation of each subgraph is tested. If a relation is redundant (either partially or totally), the removal of other relations does not affect its removal because there always exists an alternative combination of relations to enforce consistency in the same way.

7.2.1.2. Trees and the Decomposition Procedure

The previous section showed that applying decomposition methods on a consistency specification results in finding the optimal tree decomposition. However, these methods are only defined in accordance with the theoretical framework for consistency defined in Section 3.1. Providing an implementation of these decomposition methods, i.e. a decomposition procedure, introduces fundamental limitations in the generation of an optimal decomposition. We now aim to evaluate changes that occur in the finding of optimal decompositions when considering the implementation of decomposition methods.

On the plus side, the implementation of the decomposition procedure resembles for the most part the algorithm described with decomposition methods. Therefore, it is possible to find invariants and common features between the (functionally optimal) theoretical approach and its implementation. First, graph theory is used in both approaches. The consistency relation graph of decomposition methods is transformed into a metagraph, i.e. a hypergraph that embeds both consistency relations and consistency rules. As a result, algorithms pointed out in Section 4.2.4.2 are still used in the implementation of the decomposition procedure. For example, the computation of connected components is a prerequisite for both approaches. Similarly, the search for alternative combinations of consistency relations in the consistency relation graph is also interpreted as a search for alternative combinations of consistency rules in the metagraph. Both are regarded as finding paths in graphs. For the sake of convenience and ease of implementation, the problem of finding a path in the metagraph is reduced to the problem of finding a cycle in its dual. These common features imply that the metagraph is structurally as powerful as the consistency relation graph, in the sense that the logic behind redundancy detection uses similar data structures and similar algorithms.

However, some properties of decomposition methods are difficult (or even impossible in the general case) to implement in the decomposition procedure. For the most part, this relates to the comparison between consistent instances according to consistency relations and alternative combinations of consistency relations. As explained in Section 5.3.4.1, the definition of partially and totally redundant consistency relations involves the comparison of (possibly infinite) sets of instances. Because such a comparison is programmatically impossible, we rather compare intensional definitions of sets of instances. These definitions are written as OCL expressions, so the decomposition procedure has to perform a static analysis of OCL expressions. We showed in Section 6.1.3 that it is impossible in general to ensure that an OCL expression matches a certain specification.

This represents a serious limitation to the detection of redundancy, as there exist comparisons between consistency rules and alternative combinations of consistency rules that cannot be performed. We also showed in Section 6.4.8 that such comparisons involve OCL constructs that frequently occur in practice. As a result, a consistency specification can include consistency rules that the decomposition procedure is unable to remove because of the undecidability of OCL. This also means that the decomposition procedure does not always find the optimal decomposition. It is impossible to draw up an accurate and exhaustive list of OCL constructs that prevent the decomposition procedure to perform optimally. The reason for this is that current state-of-the-art SMT solvers (such as the one embedded in the procedure) use many heuristics. Consequently, the line between what is decidable and what is not is not always clear. However, some OCL constructs significantly reduce the chances of detecting redundancy in a rule. For example, OCL operations on collections are often translated using quantifiers (such as the *includes* operation that uses an existential quantifier to check the existence of an element). Quantifiers in first-order formulae make decisions harder, so it is usually better to stick to *quantifier-free fragments* (QFF) of first-order logic when possible. Also, if the consistency rule to analyze includes many QVT-R variables, these variables become existentially quantified as explained in Section 5.4.4.1. The number of quantified variables is also a criterion of complexity to find out if the translated Horn clause is valid.

The other limitation in finding optimal decompositions is of a different nature. Not all constructs of QVT-R and OCL are currently supported in the decomposition procedure. Limitations make it easier to analyze and to translate consistency specifications. In Chapter 6, we gave an overview of features of OCL that are supported or not in the decomposition procedure. These limitations are an obstacle to the optimality of the procedure in the sense that valid QVT-R consistency specifications with a tree decomposition may result in errors because of unsupported constructs instead of the expected decomposition.

Altogether, decomposition methods that operate on consistency relation graphs are able to find the optimal decomposition of a consistency specification by construction. Moreover, the merge of consistency relations and consistency rules leads to a data structure called the *metagraph* that is structurally as powerful as consistency relation graphs. However, the use of the metagraph in the context of an implementation of the decomposition procedure brings two limitations. The first limitation is that it is impossible to detect redundancy in general due to the undecidability of OCL and QVT-R. The second limitation is that we introduced intentional limitations in consistency specifications to analyze. These limitations make it impossible for the decomposition procedure to return the optimal decomposition in general. The evaluation of the applicability of the procedure provides further insights into the consequences of these limitations.

7.2.2. Unaltered Consistency Specifications

Whether a tree-like specification can be returned or not, the decomposition procedure must meet some important criteria in order to be integrated into a consistency preservation process. These criteria were described as *formal properties* of the decomposition procedure. There are two of them: *conservativeness*, ensuring that the decomposition procedure does

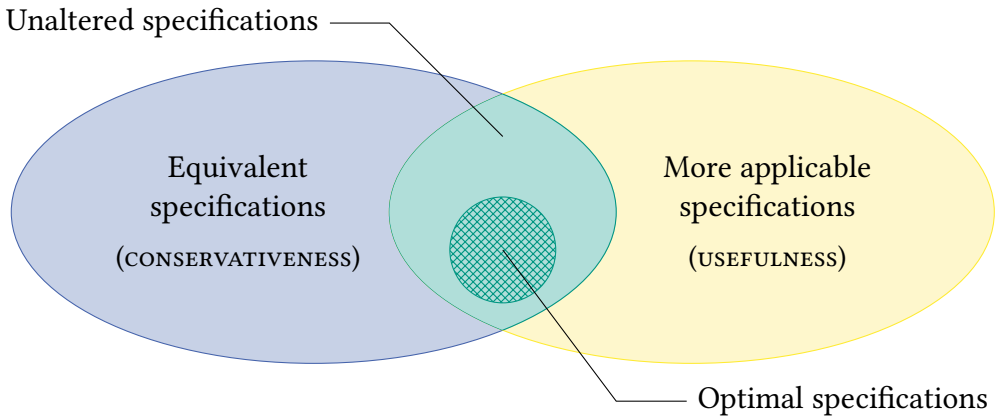


Figure 7.1.: Classification of resulting specifications

not alter the consistency specification (i.e. consistent models are the same before and after decomposition) and *usefulness*, ensuring that the resulting specification is in some way easier to apply that the input specification.

The decomposition procedure being an optimization technique, these properties ensure that even if the result is not an optimal solution (i.e. a tree of consistency relations), it is at least a feasible solution. Figure 7.1 depicts the different categories to which a specification resulting from the decomposition procedure can belong. In the context of this thesis, we aim to provide unaltered specifications, i.e. consistency specifications that may not be optimal but that meet at least the two formal properties identified above.

7.2.2.1. Equivalence of Input and Output Specifications

We now evaluate the fulfillment of *conservativeness* by the implementation of the decomposition procedure. Intuitively, the procedure is conservative if and only if each QVT-R construct that participate in the definition of the consistency specification is somehow integrated into the decomposition procedure and used to detect redundancy. In our implementation, this is achieved in two ways. First, according to Section 5.3.2, all input transformations and relations are processed, so that each relation that may have been visited during the execution of QVT-R transformations is integrated into the metagraph. Thus, no important relation is left out.

Second, consistency rules are defined inside a QVT-R relation by using variables. In our implementation, we provide a support of all QVT-R constructs in which there are variables or metamodel elements: domain patterns, preconditions and invariants. This means that each template expression is integrated into the metagraph. In particular, property template items become labels of meta-edges, preconditions are checked during the construction of the metagraph and invariants become additional conditions for QVT-R variables. We described the exact processing of these constructs in Section 5.4.4.

The structure of the metagraph ensures that all metamodel elements bound by the same QVT-R variable are grouped into the same meta-edge, so that the potential consistency rule between them must be fulfilled. In this way, the computation of connected components cannot separate them and maybe break a consistency rule. After the construction of the

metagraph, all interrelated metamodel elements belong to the same meta-edge and each OCL expression appears (translated) in the metagraph.

Afterwards, the decomposition procedure tries to delete as many consistency rules as possible by removing their associated meta-edges. For the procedure to be conservative, we ensured that the removal of a meta-edge is only performed if there exists an alternative combination of meta-edges to replace it. This is the whole purpose of the translation of expressions into first-order logic: validity proofs are required for the detection of redundancy. Therefore, conservativeness relies on the correctness of the SMT solver and the correctness of the translation. The former can be reasonably assumed, whereas the latter was backed with tests during the development of prototype. Note that the translation from OCL to Z3 is not exhaustive. As a result, a choice of implemented operations and data structures was made to provide a correct and complete enough support of OCL to develop a proof of concept of the decomposition procedure. Moreover, we deliberately left out the *undefinedness* of OCL, i.e. the fact that OCL expressions may evaluate to *undefined*, so that OCL can be mapped to a two-valued logic. This choice will be discussed in the applicability of our approach.

7.2.2.2. Usefulness of the Output Specification

We now evaluate the fulfillment of *usefulness* by the implementation of the decomposition procedure. Intuitively, an output specification is said to be useful if all its consistency relations are derived from input consistency relations. That is, the procedure used the input specification to perform the decomposition and did not add "artificial" relations.

This property shows up quite easily in the implementation of the decomposition procedure. First, no meta-edge is added to the metagraph after the initial phase of construction. Consequently, there are only two actions performed on the metagraph after its construction: the computation of connected components (which leaves the number of meta-edges unchanged) and the removal of meta-edges. At the end of the procedure, the output specification is made up of the same consistency rules as the input specification (minus those removed because of redundancy). The worst case happens where the procedure did not find any meta-edge to remove. In such a case, the input and the output specifications are equal. We formalized this as the *trivial* decomposition in Section 4.3.2, which also meets the property of usefulness.

7.3. Applicability

The applicability of the decomposition procedure assesses whether the decomposition procedure proposed in this thesis is likely to be applied in practice, i.e. to be used as is to optimize networks of consistency relations.

To the best of our knowledge, there exists no similar algorithm that optimizes consistency specifications represented as QVT-R transformations and could serve as a reference to evaluate the applicability of our approach. As a result, the decomposition procedure is executed on scenarios verifying specific aspects of OCL and QVT-R for which the existence a tree decomposition has been determined in advance.

As a preamble, it should be noted that the applicability of the decomposition procedure is constrained by the choice of technologies for its implementation. Results presented in this section were obtained by running an implementation built as follows. First, the support of the QVT-R standard is provided by QVTd[1], an Eclipse project providing a partial implementation of QVT-C and QVT-R that is still under development at the time of writing of this thesis. Second, the support of Essential OCL is provided by Eclipse OCL[2]. Third, the support of Ecore is also part of the core EMF framework[3]. Regarding tools external to model-driven engineering, we use Z3[4] to check the satisfiability of translated Horn clauses. Finally, we use some graph algorithms of the JGraphT library[5].

Application is evaluated as follows. First, we describe our example scenarios, i.e. consistency specifications that serve as inputs for the prototype of the decomposition procedure. In particular, we describe the way these scenarios were built. Then, these consistency specifications are used as inputs for the decomposition procedure. We set up a metric to explain what example scenarios evaluate and we discuss execution results.

7.3.1. Example Scenarios

Example scenarios were developed according to QVT-R and OCL constructs, i.e. to evaluate the encoding and the translation of specific aspects of consistency specifications. Not all consistency specifications have an optimal decomposition, so that scenarios also evaluate the detection of incompatibilities. Table 7.1 shows all 19 scenarios developed to assess the applicability of the decomposition procedure. They are primarily composed of three or four metamodels, primitive datatypes and operations presented in Chapter 6.

Examples that should result in a tree, i.e. optimal, decomposition after the execution of the procedure are said to be *tree-based*. Some other examples were developed with explicit contradictory relations to check the conservativeness of the procedure. Scenarios are only made up of supported QVT-R and OCL constructs. For scenarios including unsupported constructs, the prototype would raise an exception to avoid the construction of incorrect metagraphs. The distinction between unsupported and non-decomposable consistency specifications is further explained in Section 7.4.2.1.

7.3.2. Execution Results

7.3.2.1. Expected Decompositions

The execution of a consistency specification by the prototype of the decomposition procedure provides a list of independent consistency subgraphs that correspond to connected components in the metagraph as well as a list of removed consistency rules for each connected component. Expected results are as follows.

[1] https://wiki.eclipse.org/QVTd
[2] https://wiki.eclipse.org/OCL
[3] https://wiki.eclipse.org/Ecore
[4] https://z3prover.github.io
[5] https://jgrapht.org

	Description of the scenario	Tree-based
1	Three equal `String` attributes of three metamodels.	✓
2	Six equal `String` attributes of three metamodels.	✓
3	Concatenation of two `String` attributes.	✓
4	Double concatenation of four `String` attributes.	✓
5	Substring in a `String` attribute.	✓
6	Substring in a `String` attribute with precondition.	✓
7	Precondition with all primitive datatypes.	✓
8	Absolute value of an `Integer` attribute with precondition.	✓
9	Transitive equality for three `Integer` attributes.	✓
10	Inequalities for three `Integer` attributes.	✓
11	Contradictory equalities for three `Integer` attributes.	✗
12	Contradictory inequalities for three `Integer` attributes.	✗
13	Constant property template items.	✓
14	Linear equations with three `Integer` attributes.	✓
15	Contradictory linear equations for three `Integer` attributes.	✗
16	Emptiness of various OCL sequence and set literals.	✗
17	Equal `String` attributes for four metamodels.	✓
18	Transitive inclusions in sequences.	✓
19	Comparison of role names in three metamodels.	✓

Table 7.1.: Example scenarios considered in the evaluation of applicability

- *For tree-based specifications.* The expected result is the optimal decomposition, i.e. the procedure must remove as many meta-edges as possible and return a decomposition only composed of trees.

- *For non-tree-based specifications.* These specifications include contradictory relations. The expected result is a decomposition in which contradictory consistency relations have not been removed.

The interpretation of execution results depends on the number of scenarios leading to the expecting result. For scenarios that do not lead to the expected result, we also check if the decomposition size (see Section 4.3.2) was improved and the reasons of failure.

7.3.2.2. Results Obtained

On 19 example scenarios of Table 7.1, there are 16 scenarios providing the expected result and 3 inaccurate scenarios results. In detail, problematic scenarios are:

- *Scenario 8.* This scenario should have removed one meta-edge because of a transitive relation involving absolute values. The problem is the precondition of one of the relation. To test the implementation of operations related to set literals, the precondition checks the inclusion of an element in the intersection of two set literals. The SMT solver returned `UNKNOWN` so the relation was not considered.

- *Scenario 18.* This scenario checks that for three sets A, B and C, $A \subset B$ and $B \subset C$ imply $A \subset C$. The SMT solver returned `UNKNOWN` instead of `UNSAT` so one meta-edge was not removed. The translation of inclusion involves quantifiers.

- *Scenario 19.* This scenario checks the same principle as scenario 18 with local invariants. Role names are used to compare sets of metaclasses with equivalent identifiers. The translated Horn clause includes many quantifiers and uninterpreted functions. The SMT solver returned `UNKNOWN` instead of `UNSAT`.

As a result, basic operations on primitive datatypes are easily processed with the decomposition procedure. This includes non-trivial constraints that involve several integer equations or string operations. However, more complex operations and structures involving many quantifiers often prevent the SMT solver to prove the unsatisfiability of translated Horn clauses. In the context of consistency specifications, this mainly relates to role names and collections. This is not a surprising result in the sense that quantifiers severely restrict the decidability of formulae. Moreover, no strategies or tactics were used to find adapted heuristics for quantifiers [DP13]. At this point, the decomposition procedure is therefore better able to manage specifications whose consistency rules are made up of attributes and primitive datatypes.

7.3.3. Threats to Validity

The validity of results presented in the evaluation of the applicability of the decomposition procedure may be subject to changes. Four threats to validity are presented below.

- *Representative sample of specifications.* Example scenarios presented in this section are rather artificial in the sense that they were developed to evaluate specific aspects of the procedure. Consistency specifications developed here may not be representative of specifications written by transformation developers.
- *Size of specifications.* Example scenarios presented in this section were all made up of a small number of metamodels and consistency relations. The response time of the decomposition procedure depending on the size of the consistency relation graph was not tested. Paton's algorithm has a time complexity $O(V^3)$ where V is the number of vertices in the dual graph. In practice, there should be no difference for specifications with a few hundred consistency rules.
- *Evolution of standards.* The prototype of the procedure relies on languages such as QVT-R whose implementation is still under development. The clarification of the QVT standard as well as the evolution of QVTd may affect the results of the decomposition procedure in the future.
- *Evolution of solvers.* Similarly, the procedure relies on the Z3 SMT solver. Satisfiability modulo theories is an active field of research. Future updates of the solver and of SMT-LIB may change the input format and affect the translation of OCL expressions.

7.4. Discussion and Further Evaluation

In the two previous sections, we discussed the functional correctness and the applicability of the decomposition procedure. These approaches allow for a more general discussion of benefits and limitations of using the decomposition procedure.

7.4.1. Benefits

In light of the evaluation conducted in the two previous sections, there are several factors according to which the decomposition procedure is of benefit. First, we discuss the role of the procedure in the detection of incompatibilities. Then, the ease of integration of the decomposition procedure is highlighted.

7.4.1.1. Detection of Incompatibilities

The purpose of the decomposition procedure is to help to improve the *applicability* of consistency specifications. Applicability is defined as the usability of consistency specifications to preserve multi-model consistency. In practice, the applicability in networks of consistency relations is threatened by interoperability issues. In Section 4.1.2, we presented four challenges related to interoperability issues. The result showed that compatibility

was the most important challenge to solve. In other words, applicability of consistency specifications is mostly driven by the compatibility of their consistency relations.

The decomposition procedure is an optimization technique that aims to produce a tree-based specification by removing as many consistency relations as possible without altering the specification. Ideally, all consistency relation graphs are trees or sets of trees, because it inherently ensures compatibility. Ensuring that all consistency relations are compatible with each other amounts to showing the following property. There should exists at least one consistent model after the execution of all consistency relations. Suppose that there exist two combinations of relations C_1 and C_2 between two metamodels. If there is no model that is consistent according to both C_1 and C_2, then there is an *incompatibility* and there are contradictory consistency relations among C_1 and C_2. Since there are never two different paths between two vertices of a tree, tree-based specifications cannot include contradictory consistency relations.

Because of conservativeness, the procedure does not alter its input specification. This means that the procedure does not directly improve compatibility among relations of the specification. However, the resulting decomposition can indirectly reduce the risk of incompatibilities. For example, take decompositions $\mathcal{D}_1$ and $\mathcal{D}_2$ of Figure 4.7.

- *Optimal case.* Decomposition $\mathcal{D}_1$ is optimal because it is a disjoint union of trees. What could have been contradictory relations in the initial specification were only redundant or independent consistency relations. There cannot exist contradictory consistency relations in the specification. As a result, the procedure has reduced the risk of incompatibilities to zero, thus indirectly ensuring compatibility.
- *Non-optimal case.* Decomposition $\mathcal{D}_2$ is not optimal because there still exists several combinations of consistency relations between M_4 and M_5. This means that the decomposition procedure was unable to find redundancy and to reduce the subgraph to a tree.

There may be two reasons why the resulting decomposition is not only made up of trees. First, the specification can include contradictory consistency relations. In this case, it is impossible to prove that one consistency relation can be replaced by the others. Second, a consistency relation is indeed redundant but the decomposition could not prove it due to practical limitations of SMT solving on certain Horn clauses. As a result, the transformation developer must analyze remaining cycles in consistency subgraphs to modify the specification if necessary. Put another way, the decomposition procedure does not repair contradictory relations. It optimizes specifications so that possibly contradictory relations become easier to find.

7.4.1.2. Ease of Integration

The other benefit of the decomposition procedure is the ease of integrating the procedure in a consistency preservation process. There are mainly two reasons for this: automation and conservativeness. First, the procedure is fully automated, especially thanks to the use of an automated theorem prover. The benefit of automation is that the decomposition procedure

could be used as a watchdog before executing consistency preservation mechanisms. Each update of a consistency specification may introduce new incompatibilities. Therefore, using the decomposition procedure as soon as the specification is updated can allow potential incompatibilities to be reported to the transformation developer.

Second, the decomposition procedure is conservative. This means that the resulting consistency specification is equivalent to the input consistency specification. In other words, the input specification can be replaced by the output specification in the consistency preservation process. Regarding the usefulness property, we defined in Section 4.3.2 the decomposition size, a measure of the optimality of a decomposition. This metric can be used by the transformation developer to check whether the risk of incompatibilities has grown after an update of the specification.

7.4.2. Limitations

The decomposition procedure is an optimization technique that cannot alter consistency specifications. In the worst case, the procedure is unable to perform a decomposition. The resulting specification is then identical to the input specification. This is an important requirement indicating that the procedure cannot harm the consistency preservation process. However, there are limitations to the use of the decomposition procedure.

7.4.2.1. Partial Support of Specifications

Not all consistency specifications are supported by the procedure. More precisely, the partial support of consistency specifications creates two types of restrictions:

- *Unsupported specifications.* There are specifications which cannot be interpreted by the decomposition procedure. The reason for this is that not all QVT-R and OCL constructs have been encoded in the metagraph or translated into Z3. Regarding QVT-R, these constructs are collection template expressions, metamodel elements in conditions, opposite roles and variable declarations in property template items. Regarding OCL, unimplemented constructs are those that are not provided in Chapter 6. Such specifications raise exceptions in the prototype. Otherwise, consistency could be altered. This type of restriction is specific to the procedure, i.e. further research on decomposition can help to overcome these restrictions.

- *Non-decomposable specifications.* Even if all constructs of a consistency specification can be encoded into a metagraph and translated into Z3, the decomposition is not always optimal. The reason for this is that SMT solvers have theoretical and practical limitations too. This type of restriction is external to the procedure.

7.4.2.2. Time Complexity of the Procedure

The time complexity of the decomposition procedure has not been formally established. Using only a consistency specification, it is hard to infer the size of the resulting metagraph, which is itself used for the time complexity of graph algorithms. Moreover, the procedure embeds a SMT solver as a black box. The running time of Z3 depends on many heuristics

and is very unpredictable. As a result, it is difficult to know how long it will take to decompose a given specification.

No example scenarios defined in Section 7.3.1 resulted in a time-consuming decomposition. It is noteworthy, however, that the decomposition procedure may require a lot of time in some very large consistency specifications encoded as very large metagraphs with specific topologies (e.g. graphs that contain exponentially many simple cycles).

7.4.3. Further Evaluation

7.4.3.1. Simulated Redundancy

A more advanced strategy to perform an empirical evaluation of the prototype of the decomposition procedure is to introduce redundancy in consistency specifications and to assess whether the procedure is able to detect it. More precisely, the procedure is applied on input specifications with rather sparse topologies.

Automatic redundancy can be introduced by *composing* transformations and consistency relations. For example, the composition of two transformations $A \leftrightarrow B$ and $B \leftrightarrow C$ leads to the creation of another transformation $A \leftrightarrow C$. As a result, the consistency relation graph is more dense. The evaluation then consists in verifying that the procedure is able to decompose composed consistency relations. An appropriate metric in this context is the number of artificial composed relations that were replaced by the procedure. Related work on the composition of transformations was presented in Section 8.1.3.

7.4.3.2. Real Case Scenarios

Another strategy to provide a more realistic assessment of the applicability of the decomposition procedure is to use consistency specifications that represent real case scenarios as inputs of the procedure. The current evaluation scenarios are mainly based on OCL and QVT-R constructs to measure the scope of specifications that can be processed.

As there currently exists no implementation that fully supports the QVT-R language, consistency checking and consistency enforcement by means of QVT-R transformations are still scarce. Imperative and hybrid model transformation languages tend to provide a better support than declarative languages. As a result, there are very few usable QVT-R specifications. To illustrate the concepts of the language, the QVT standard includes *UML2RDBMS*, a nontrivial example of QVT-R relations that establishes a correspondence between UML and a relational database management system [Obj16b].

8. Related Work

In this thesis, we introduced an optimization technique for consistency specifications in the context of multi-model consistency preservation. This technique relies on the static analysis of a transformation language, QVT-R, and a declarative specification language, OCL. It is also based on the use of an automated theorem prover (or more precisely here a satisfiability modulo theories (SMT) solver) to perform the detection of independent subsets of consistency relations.

Consequently, this thesis is at the intersection of consistency preservation and formal verification in model-driven software development. This chapter is an overview of the research work carried out on these topics.

The first section reviews the main approaches that address the problem of model consistency. The second section provides an overall picture of the use of formal methods to reason about properties of transformation languages, including QVT-R.

8.1. Model Consistency Preservation

8.1.1. Approaches for Consistency

The concept of consistency is widely used in computer science. Being rather vague, the term "consistency" has been researched from many perspectives and in many areas. Oldest occurrences mainly relate consistency in database systems to discuss concepts such as transactions and locks [Esw+76].

In software engineering, consistency preservation is an inherent theme in model-driven engineering. Since the beginning of domain-specific languages, needs for consistency were identified [Fin00]. Similarly, consistency among UML models was discussed early [MVS][DMW05]. More generally, lacks of consistency, namely *inconsistencies*, were evaluated by Nuseibeh et al. [NER00] and by Spanoudakis et al. [SZ01].

Aside from the graph formalism used in this thesis, there are many formalisms to model consistency. Macedo et al. proposed a review of these formalisms arranged by features [MJC17]. The most frequent are introduced here.

8.1.1.1. Triple Graph Grammars

Triple Graph Grammars (TGGs) is a concept of graph transformation introduced by Schürr [Sch94]. TGGs serve as a formal way to describe model transformations. The main idea behind TGGs is to represent models as graphs and model transformation as graph rewriting. In each graph rewrite rule, there are three graphs. The first one is the *pattern* graph, i.e. the one to be replaced. The second one is the *replacement* graph. It provides patterns to

perform the update. The third one, called the *gluing* graph, is an interface to describe the common elements in the first two graphs.

There exist several tools using TGGs to perform *model synchronisation*, i.e. to update models so that consistency is restored [Leb+14]. Note that the specification of the QVT-R language was strongly influenced by triple graph grammars [GK07]. This can be observed in the graphical representation of QVT-R relations.

8.1.1.2. Bidirectional Transformations

Bidirectional transformations (bx) are another formalism for consistency preservation. They are usually defined as a mechanism for maintaining consistency between two (or more) related sources of information [Che+14]. Bx are very generic, in that they have already been applied in many research areas [Cza+09]. In model-driven engineering, bx can be used to codify the process of restoring consistency if a model is updated [Ste08]. Bidirectionality means that during the execution of the transformation, all models can be source models, target models or both. They lay the foundations for this thesis, as transformations defined to restore consistency in the consistency relation graph are bx.

The study of bx gave rise to many theoretical frameworks and variants [DM14]. For example in a *state-based* approach of bx, states of models are compared and transformations are performed according this comparison. In a *delta-based* approach of bx, only changes between models are monitored and transformations are regarded as sequences of changes. Another classification of bx uses symmetry: a bx is symmetric if each model contains information that is not present in other models[Dis+11].

As stated by Stevens [Ste10], QVT-R can be considered as a language to write bidirectional model transformations, making QVT-R a language that can be both formalized by bx and TGGs. The connections between bidirectional transformations and TGGs have been researched by Königs [Kön05].

8.1.1.3. Various Approaches

While previous approaches are rather theoretical, several languages and tools have been implemented to allow consistency preservation in practice.

One of the transformation languages with the best support is the *ATLAS Transformation Language* (ATL) [Jou+08]. ATL is unidirectional. Xiong et al. described a model synchronisation system using ATL [Xio+07]. JTL is another transformation language that aims to support consistency preservation [Cic+10]. JTL is bidirectional and delta-based.

Viatra (currently Viatra3) is a model transformation platform providing a transformation-based verification and validation of models [Ber+15]. Among other features, Viatra can automatically check consistency for a set of models. It is event-driven, i.e. actions on models are triggered when models are changed, and reactive, i.e. it relies on concepts of reactive programming.

Finally, Vitruvius is an approach for *view-centric* model-driven software development [KBL13]. It consists of *flexible views* that are created dynamically to share specific aspects of models with developers. Vitruvius offers the possibility of maintaining consistency between models through incremental transformations. There are two languages for

consistency specifications: the Mappings (declarative, bidirectional) and the Reactions (imperative, unidirectional) languages defined by Kramer [Kra17].

The Mappings language being declarative and bidirectional, it is the most suitable for comparison with QVT-R. As stated by Kramer, the main difference between Mappings and QVT-R is the way consistency is enforced during execution. In QVT-R, a transformation invoked for enforcement is executed in a particular direction. Therefore, the semantics of the specification depends on the direction of execution. Thanks to special constructs (e.g. single-sided conditions), this is not the case in the Mappings language.

8.1.2. Multi-Model Consistency Preservation

Consistency preservation is often researched for pairs of models. *Multi-model consistency preservation* focuses on scenarios where the consistency of multiple models must be preserved. In their classification of model repair approaches, Macedo et al. also survey the multi-model case [MJC17].

There are two general trends in multi-model consistency preservation. The generalization of concepts of consistency preservation is achieved in two ways. Either concepts (such as bx or TGGs) are left unchanged but combined (sometimes also composed) or they are extended to operate on many models at the same time. The former is referred to as *pairwise multi-model consistency*, the latter as *generic multi-model consistency*

8.1.2.1. Pairwise Multi-Model Consistency

The first idea to achieve multi-model consistency preservation is to use well-researched concepts of consistency preservation such as bidirectional transformations or triple graph grammars and to use them to define consistency for a set of metamodels. Concerning bx, Stevens proved that under some reasonable conditions, multiary bx, i.e. bx maintaining consistency between an arbitrary number of metamodels, can be replaced by a set of binary bx [Ste17a]. This result is the reason why consistency relation graphs in this thesis are networks of metamodels with binary consistency relations.

Repercussions of this approach on consistency restoration in megamodels have been researched by Stevens [Ste18]. Klare also identified challenges of networks of consistency relations based on binary bx [Kla18]. The particular case of interoperability issues was surveyed more in detail by Klare et al. [Kla+19].

A similar approach called *multi-model consistency management* and based on sets of binary bx was presented by Stünkel and al. [Stü+18]. This approach uses concepts from the Diagram Predicate Framework (DPF), a formalisation of model-driven engineering using diagrams of category theory [Rut10].

8.1.2.2. Generic Multi-Model Consistency

Regarding bidirectional transformations first, theoretical aspects of a delta-based approach of multiary bx were researched by Disking et al. [DKL18]. By accepting an arbitrary number of domains inside relations, QVT-R can represent multiary bidirectional transformations.

However, Macedo et al. have shown that ambiguities and omissions of the QVT standard make this use difficult [MCP14].

Regarding Triple Graph Grammars then, Königs and Schurr introduced an extension to TGGs called *Multi-Graph Grammars* (MGGs) [KS06]. In MGGs, the gluing graph can relate an arbitrary number of metamodel graphs. Another extension to TGGs was proposed by Trollmann and Albayrak [TA16].

Whether it is for multiary bx or generalizations of TGGs, these new approaches are most often theoretical and do not discuss interoperability issues. The lack of research on applicability makes these formalisms less suitable to explore the optimization of consistency specifications (as investigated in this thesis).

8.1.3. Model Transformation Decomposition and Composition

Although there is little research on the decomposition of consistency relations or model transformations, there are results on the composition of model transformations. For example, Wagelaar et al. discussed the semantics of composed model transformations with ATL examples [Wag+11]. Composition aims to foster the development of smaller and reusable transformations. As explained by Belaunde, this type of composition is called *external* because transformations are reused as a whole and regarded as black-boxes.

Conversely, *internal* composition allows the modification of model transformations so that they can be composed more easily. For example, Etien and al. discuss the combination of independent model transformations using an `Extend` operator that modifies inputs and outputs of transformations to make them compatible [Eti+10].

Two existing approaches share similarities with the concept of decomposition defined in this thesis. First, *factorization* finds common functionality shared between multiple transformations and extracts it in a new base transformation [CM08]. Non-common functionality is implemented by other transformations deriving the base transformation. Second, *modularization* is defined by Kurtev et al. as a way to foster extensibility and reusability in model transformations by considering transformation language constructs as modules [KBJ07]. For example, in a transformation rule, the left-hand side and the right-hand side of the rule can be separated to form two modules that can be reused as *modular units* and combined.

8.2. Formalization of QVT-R

Standardized for the first time in 2008 by the Object Management Group, QVT-R is one of the main fully declarative transformation languages in model-driven engineering [Obj16b]. The declarative nature of a programming language often implies an absence of side effects. Consequently, these languages are well suited to a mathematical formalization and QVT-R is no exception. In this thesis, main QVT-R constructs have been encoded in a hypergraph. Many other formalisms have been researched.

A reason behind the multiple attempts to formalize QVT-R is the existence of inconsistencies in the standard, especially regarding consistency enforcement. An overview of these inconsistencies was reported by Stevens [Ste10].

First, QVT-R transformations have been mapped to modal μ-calculus, an extension of propositional modal logic, and game theory concepts to formalize semantics of the language regarding the `checkonly` and `enforce` execution modes [BS13]. This aims to clarify the QVT-R standard. In the same vein, a system called Maude using rewriting logic, another kind of logic, was used as a transformation engine to execute QVT-R [BCR06].

Another attempt to formalize QVT-R uses coloured Petri nets (CPN) [GL14]. A Petri net is a modeling language to describe distributed systems with formal execution semantics. Applied to QVT-R, CPN can help to debug transformations. There are also approaches using algebraic formalisms. First, a formalization approach to resolve ambiguities of the `checkonly` execution mode of QVT-R has been researched using category theory [GL12]. Second, semantics of both QVT-O and QVT-R have been described in the formalization of an "hybrid" QVT transformation by Giandini et al. [GPP09].

8.3. Formal Techniques for Transformation Languages

This thesis involves the use of formal techniques to reason about consistency specifications. More precisely, an automated theorem prover is used as a black box inside the decomposition procedure to determine if a consistency rule can be removed from the specification. The use of formal techniques with model transformations is frequent.

In fact, formal techniques are often used when there is a need of automation. As stated by Hidaka, a purpose of model-driven engineering is to achieve "*some automated goals in the production, maintenance or operation of software intensive systems*" [HID+13]. When challenges posed by automation depend on content analysis of models, specifications, etc., formal techniques offer a good support.

This section surveys the use of formal methods (close to the one used in this thesis) in the context of model-driven software development. Three types of formal methods are discussed: automated techniques, interactive techniques and model finding.

8.3.1. Automated Techniques

Automated formal techniques are techniques that do not require a human intervention during the execution. The SMT solver used in the decomposition procedure to reason about QVT-R transformations belongs to this category.

The need for formal techniques often comes with the analysis of transformations. Cuadrado et al. implemented a method to perform a static analysis of model transformations [CGL16]. The method is illustrated with ATL and uses the USE Validator, a constraint solver for models. In the same vein, Büttner et al. proposed the use of a SMT solver to verify ATL transformations [BEC12].

Regarding OCL, Kuhlmann et al. discussed the validation of OCL models using a SAT solver [KHG11]. EMF models enhanced with OCL were also subject to verification using constraint logic programming [Gon+12]. Finally, Cabot et al. also proposed to translate the verification of OCL invariants and QVT-R constructs as a constraint satisfaction problem [Cab+10].

8.3.2. Interactive Techniques

Interactive formal techniques, also called semi-automated formal techniques, are techniques that act as assistants. For example, proof assistants help to produce proofs in collaboration with humans.

An encoding of QVT-R transformations using the Coq proof assistant was implemented by Rentschler et al. [Ren15]. Another proof assistant, HOL, was used to implement HOL-OCL, a formal proof environment for UML and OCL [BW08]. The integration of a proof assistant in model-driven engineering tools can be further extended. For example, Cheng et al. designed CoqTL, a transformation language using the syntax of the specification language of Coq to avoid the usual phase of translation between the specification language and the interactive theorem prover [CTD19].

8.3.3. Model Finding

A less formal but widely used technique in model-driven engineering is *model finding*. Rather than proving properties on models, the role of model finding is to find models that are counterexamples to assertions made. Model finding is sometimes regarded as a *lightweight formal method*.

The aim of model finding is to provide a high-level and automated analysis of models in opposition to interactive techniques that are only semi-automated and to automated techniques that are difficult to embed. The compromise of model finding is that the tool can find models but if it does not find them, it cannot prove that they do not exist.

Both ATL and QVT-R were implemented using the model finding tool Alloy to repair inconsistencies [MC13]; [MC16]. The resulting transformation-based tool for QVT-R is called *Echo* [MGC13].

9. Conclusion and Future Work

9.1. Conclusion

Consistency is a fundamental requirement in model-driven engineering. Whether it is for distributed systems, databases or model-based software systems among others, consistency problems arise when data is replicated, updated and shared. In model-driven engineering, consistency ensures that models can operate together without risking undefined behaviours. In this thesis, we investigated the decomposition of consistency relations, an optimization technique for networks of consistency relations in the context of multi-model consistency preservation. More precisely, we formalized the idea of decomposition following the observation that some consistency relations could be replaced by combinations of other relations without altering a specification. We also explored the benefits of such an approach and provided various methods to perform a decomposition.

Consistency specifications are useful when they are applicable, i.e. when there are no contradictory consistency relations in the specification. The decomposition procedure is a relevant approach to make consistency specifications more applicable. More precisely, the procedure does not repair contradictory relations; this is a consequence of conservativeness. However, it optimizes specifications so that possiby contradictory relations become easier to find.

To turn decomposition into an applicable and concrete technique, we first showed how the QVT-R transformation language could be used to write consistency specifications. Consecutively, we designed a decomposition procedure, i.e. an implementation of decomposition methods. We also developed a proof of concept for the procedure. The development of the decomposition procedure required the use of an automated theorem prover. Therefore, we made a link between these two paradigms: we provided a translation of consistency relations into first-order formulae. We also developed an appropriate data structure (called a *metagraph*) to detect redundant relations. The use of the decomposition procedure results in the detection of independent consistency rules and in the removal of redundant consistency rules.

The evaluation of our approach demonstrated that the decomposition procedure is generally beneficial. The reason for this is that it never alters the consistency specification and, in the worst case, only returns a trivial decomposition. Each consistency rule that turns out to be independent or redundant is a useful information for the transformation developer. We also surveyed the applicability of the approach to assess whether the implementation of the procedure was bringing the expected benefits. The result is that the implementation provides good results regarding consistency rules between metaclass attributes but still needs improvements regarding collections of objects. Some ideas to overcome current limitations were also investigated. The decomposition procedure is

ultimately a solution with strong potential to participate in the achievement of multi-model consistency.

9.2. Future Work

9.2.1. Extension to Other Constructs

The scope of the decomposition procedure presented in this thesis is limited, in that not all constructs of languages used are supported. This applies to QVT-R constructs (e.g. collection template expressions, keys or conditions with metamodel elements), OCL expressions (e.g. operations and collections) and Ecore metamodel elements.

Extending the set of constructs and expressions supported by the decomposition procedure is one way to enhance its applicability to various consistency specifications. However, overcoming some limitations require significant changes in the architecture of the procedure. For example, untranslatable operations highlighted in Section 6.4.8 require the use of decision procedures of a different nature than those used in this work.

9.2.2. Extension to Other Symbolic Computation Tools

In this thesis, we chose to use an automated theorem prover to decide whether a combination of consistency rules could replace a single consistency rule. This has many advantages, the biggest one being the automatisation of the decomosition procedure. However, automated theorem provers like SMT solvers are rather low-level and unpredictable to reason about consistency relations and model transformations. This is mainly due to the difficulty of finding proofs in first-order formulae with quantifiers.

A possible extension to the current decomposition procedure is the use of another tool for symbolic computation. For example, interactive tools (like Coq[1] or Isabelle/HOL[2]) can provide more results by dropping the full automation feature and relying on humans for complex decision steps. The integration of such a tool in the process of consistency preservation could be investigated.

Ultimately, it is also possible to consider the use of several tools at a time. The decomposition procedure is built in such a way that the whole translation from OCL to Z3 is performed by a visitor class. By defining additional visitors using APIs from other tools, it is quite easy to extend the set of tools supported by the procedure. Then, redundancy is detected by verifying that tools do not return contradictory results and that at least one of them proves that the combination can replace the consistency rule to analyze.

9.2.3. Extension to Other Contexts

The choice of standards for the development of a prototype of the decomposition procedure is a limitation to its application in other contexts. Multi-model consistency preservation being a language-agnostic paradigm, imposing the use of a standard such as QVT-R

[1] `https://coq.inria.fr`
[2] `https://isabelle.in.tum.de`

to write consistency specifications prevents the use of the prototype in other contexts. However, we gave in Chapter 4 a general characterization of decomposition that only uses the consistency framework defined in Section 3.1.

It is noteworthy that the idea of a decomposition of consistency relations was first mentioned in Klare's introduction to multi-model consistency preservation [Kla18]. Following this research plan, we present two approaches to extend the decomposition procedure to other contexts. A first approach is the integration of the decomposition procedure into the Vitruvius framework [KBL13]. The framework provides the Mappings language, a bidirectional specification language for preserving consistency. Consequently, the principles of the decomposition procedure illustrated with QVT-R could be adapted to the Mappings language. This is facilitated by the fact that this language was defined according to the theoretical framework for consistency that we also used in this thesis.

The second approach mentioned by Klare is another way to achieve a tree structure of transformations using *concept metamodels*, a way to encode common concepts between different metamodels. This approach requires the construction of a tree of concept metamodels, which is a consequence of the use of the decomposition procedure. As a result, an extension of this work consists in considering its integration with other consistency preservation techniques. In the end, the decomposition procedure could be integrated as an optimization technique for consistency specifications in the consistency preservation process of Vitruvius.

Bibliography

[Abo+18] Faris Abou-Saleh et al. “Introduction to bidirectional transformations”. In: *Bidirectional Transformations*. Springer, 2018, pp. 1–28.

[Alu+95] Rajeev Alur et al. “The algorithmic analysis of hybrid systems”. In: *Theoretical computer science* 138.1 (1995), pp. 3–34.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[BKS02] Bernhard Beckert, Uwe Keller, and Peter H Schmitt. “Translating the Object Constraint Language into first-order predicate logic”. In: 2002.

[BCD05] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. “Reasoning on UML class diagrams”. In: *Artificial intelligence* 168.1-2 (2005), pp. 70–118.

[Ber+15] Gábor Bergmann et al. “Viatra 3: A reactive model transformation platform”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2015, pp. 101–110.

[BCR06] Artur Boronat, José Á Carsí, and Isidro Ramos. “Algebraic specification of a model transformation engine”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2006, pp. 262–277.

[BS13] Julian Bradfield and Perdita Stevens. “Enforcing QVT-R with mu-calculus and games”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2013, pp. 282–296.

[BW08] Achim D Brucker and Burkhart Wolff. “HOL-OCL: a formal proof environment for UML/OCL”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2008, pp. 97–100.

[BEC12] Fabian Büttner, Marina Egea, and Jordi Cabot. “On verifying ATL transformations using ‘off-the-shelf’SMT solvers”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2012, pp. 432–448.

[Cab+10] Jordi Cabot et al. “Verification and validation of declarative model-to-model transformations through invariants”. In: *Journal of Systems and Software* 83.2 (2010), pp. 283–302.

[Che+14] James Cheney et al. “Towards a Repository of Bx Examples”. In: *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.* 2014, pp. 87–91. URL: `http://ceur-ws.org/Vol-1133/paper-14.pdf`.

[CTD19] ZHENG CHENG, Massimo Tisi, and Rémi Douence. "CoqTL: A Coq DSL for Rule-Based Model Transformation". In: *Software and Systems Modeling* (2019). URL: https://hal.archives-ouvertes.fr/hal-02333564.

[Chu36] Alonzo Church. "A note on the Entscheidungsproblem". In: *The journal of symbolic logic* 1.1 (1936), pp. 40–41.

[Cic+10] Antonio Cicchetti et al. "JTL: a bidirectional and change propagating transformation language". In: *International Conference on Software Language Engineering*. Springer. 2010, pp. 183–202.

[Cle+19] Anthony Cleve et al. "Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)". In: *Dagstuhl Reports* 8.12 (2019). Ed. by Anthony Cleve et al., pp. 1–48. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.12.1. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10360.

[CGL16] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "Static analysis of model transformations". In: *IEEE Transactions on Software Engineering* 43.9 (2016), pp. 868–897.

[CM08] Jesús Sánchez Cuadrado and Jesús García Molina. "Approaches for model transformation reuse: Factorization and composition". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 168–182.

[Cza+09] Krzysztof Czarnecki et al. "Bidirectional transformations: A cross-discipline perspective". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2009, pp. 260–283.

[DMW05] Cristine R Dantas, Leonardo Gresta Paulino Murta, and Cláudia Maria Lima Werner. "Consistent evolution of UML models by automatic detection of change traces". In: *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE. 2005, pp. 144–147.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[DP13] Leonardo De Moura and Grant Olney Passmore. "The strategy challenge in SMT solving". In: *Automated Reasoning and Mathematics*. Springer, 2013, pp. 15–44.

[DC+03] Rina Dechter, David Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003.

[DF02] Rina Dechter and Daniel Frost. "Backjump-based backtracking for constraint satisfaction problems". In: *Artificial Intelligence* 136.2 (2002), pp. 147–188.

[DKL18] Zinovy Diskin, Harald König, and Mark Lawford. "Multiple Model Synchronization with Multiary Delta Lenses". In: *Fundamental Approaches to Software Engineering*. Ed. by Alessandra Russo and Andy Schürr. Cham: Springer International Publishing, 2018, pp. 21–37. ISBN: 978-3-319-89363-1.

[DM14] Zinovy Diskin and Tom Maibaum. “Category theory and model-driven engineering: From formal semantics to design patterns and beyond”. In: *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice* (2014), p. 173.

[Dis+11] Zinovy Diskin et al. “From state-to delta-based bidirectional model transformations: the symmetric case”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 304–318.

[Esw+76] Kapali P. Eswaran et al. “The notions of consistency and predicate locks in a database system”. In: *Communications of the ACM* 19.11 (1976), pp. 624–633.

[Eti+10] Anne Etien et al. “Combining independent model transformations”. In: 2010.

[FN05] Jean-Marie Favre and Tam NGuyen. “Towards a megamodel to model software evolution through transformations”. In: *Electronic Notes in Theoretical Computer Science* 127.3 (2005), pp. 59–74.

[Fin00] Anthony Finkelstein. “A foolish consistency: Technical challenges in consistency management”. In: *International Conference on Database and Expert Systems Applications*. Springer. 2000, pp. 1–5.

[Für+09] Simon Fürst et al. “AUTOSAR–A Worldwide Standard is on the Road”. In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 62. 2009, p. 5.

[GPP09] Roxana S Giandini, Claudia Pons, and Gabriela Pérez. “A two-level formal semantics for the QVT language.” In: *CIbSE* 9 (2009), pp. 73–86.

[Gib69] Norman E Gibbs. “A cycle generation algorithm for finite undirected linear graphs”. In: *Journal of the ACM (JACM)* 16.4 (1969), pp. 564–568.

[Gon+12] Carlos A González et al. “EMFtoCSP: A tool for the lightweight verification of EMF models”. In: *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE. 2012, pp. 44–50.

[GK07] Joel Greenyer and Ekkart Kindler. “Reconciling tggs with qvt”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2007, pp. 16–30.

[GL12] Esther Guerra and Juan de Lara. “An algebraic semantics for QVT-relations check-only transformations”. In: *Fundamenta Informaticae* 114.1 (2012), pp. 73–101.

[GL14] Esther Guerra and Juan de Lara. “Colouring: execution, debug and analysis of QVT-relations transformations through coloured Petri nets”. In: *Software & Systems Modeling* 13.4 (2014), pp. 1447–1472.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. “Incremental model transformation for the evolution of model-driven systems”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2006, pp. 321–335.

[HID+13] Soichiro HIDAKA et al. “Principles and Applications of Model Driven Engineering (1) History and Context of Model Driven Engineering”. In: *Computer Software* 30 (Jan. 2013). DOI: 10.11309/jssst.30.3_25.

[HT73] John Hopcroft and Robert Tarjan. “Algorithm 447: Efficient Algorithms for Graph Manipulation”. In: *Commun. ACM* 16.6 (June 1973), pp. 372–378. ISSN: 0001-0782. DOI: 10.1145/362248.362272. URL: http://doi.acm.org/10.1145/362248.362272.

[ISO11] ISO/IEC/IEEE. “Systems and software engineering – Architecture description”. In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (Jan. 2011), pp. 1–46. DOI: 10.1109/IEEESTD.2011.6129467.

[JL87] Joxan Jaffar and J-L Lassez. “Constraint logic programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1987, pp. 111–119.

[Jou+06] Frédéric Jouault et al. “ATL: a QVT-like transformation language”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pp. 719–720.

[Jou+08] Frédéric Jouault et al. “ATL: A model transformation tool”. In: *Science of computer programming* 72.1-2 (2008), pp. 31–39.

[Kla18] Heiko Klare. “Multi-model Consistency Preservation”. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’18. Copenhagen, Denmark: ACM, 2018, pp. 156–161. ISBN: 978-1-4503-5965-8. DOI: 10.1145/3270112.3275335.

[Kla+19] Heiko Klare et al. “A Categorization of Interoperability Issues in Networks of Transformations.” In: *The Journal of Object Technology* 18 (Jan. 2019), 4:1. DOI: 10.5381/jot.2019.18.3.a4.

[Kle+03] Anneke G Kleppe et al. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[Kön05] Alexander Königs. “Model transformation with triple graph grammars”. In: 2005.

[KS06] Alexander Königs and Andy Schürr. “MDI: a rule-based multi-document and tool integration approach”. In: *Software & Systems Modeling* 5.4 (2006), pp. 349–368.

[KBL13] Max E. Kramer, Erik Burger, and Michael Langhammer. “View-centric Engineering with Synchronized Heterogeneous Models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. DOI: 10.1145/2489861.2489864. URL: http://doi.acm.org/10.1145/2489861.2489864.

[Kra17] Max Emanuel Kramer. “Specification Languages for Preserving Consistency between Models of Different Languages”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284.

[KS] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer.

[Kru56] Joseph B Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem”. In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[KHG11] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. “Extensive validation of OCL models by integrating SAT solving into USE”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2011, pp. 290–306.

[KMT12] Adrian Kuhn, Gail C Murphy, and C Albert Thompson. “An exploratory study of forces and frictions affecting large-scale model-driven development”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2012, pp. 352–367.

[KBJ07] Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. “Rule-based modularization in model transformation languages illustrated with ATL”. In: *Science of computer programming* 68.3 (2007), pp. 138–154.

[LK14] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. “Model-transformation design patterns”. In: *IEEE Transactions on Software Engineering* 40.12 (2014), pp. 1224–1259.

[Leb+14] Erhan Leblebici et al. “A comparison of incremental triple graph grammar tools”. In: *Electronic Communications of the EASST* 67 (2014).

[Lif08] Vladimir Lifschitz. “What Is Answer Set Programming?.” In: *AAAI*. Vol. 8. 2008. 2008, pp. 1594–1597.

[LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: http://doi.acm.org/10.1145/321738.321743.

[Lov16] Donald W Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[MJC17] N. Macedo, T. Jorge, and A. Cunha. “A Feature-Based Classification of Model Repair Approaches”. In: *IEEE Transactions on Software Engineering* 43.7 (July 2017), pp. 615–640. DOI: 10.1109/TSE.2016.2620145.

[MCP14] Nuno Moreira Macedo, Alcino Cunha, and Hugo Pereira Pacheco. “Towards a framework for multidirectional model transformations”. In: (2014).

[MC13] Nuno Macedo and Alcino Cunha. “Implementing QVT-R bidirectional model transformations using Alloy”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2013, pp. 297–311.

[MC16] Nuno Macedo and Alcino Cunha. "Least-change bidirectional model transformation with QVT-R and ATL". In: *Software & Systems Modeling* 15.3 (July 2016), pp. 783–810. ISSN: 1619-1374. DOI: 10.1007/s10270-014-0437-x. URL: https://doi.org/10.1007/s10270-014-0437-x.

[MGC13] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. "Model repair and transformation with Echo". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 694–697.

[MD76] Prabhaker Mateti and Narsingh Deo. "On algorithms for enumerating all circuits of a graph". In: *SIAM Journal on Computing* 5.1 (1976), pp. 90–99.

[MVS] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. "Maintaining consistency between UML models with description logic tools". In:

[MV06] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142.

[MB11] Leonardo de Moura and Nikolaj Bjørner. *Z3 - a Tutorial*. 2011. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.225.8231&rep=rep1&type=pdf.

[NER00] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. "Leveraging inconsistency in software development". In: *Computer* 33.4 (2000), pp. 24–29.

[Obj16a] Object Management Group (OMG). *Meta-Object Facility (MOF) Specification, Version 2.5.1*. OMG Document Number formal/2016-11-01 (https://www.omg.org/spec/MOF/2.5.1). 2016.

[Obj16b] Object Management Group (OMG). *MOF Query/View/Transformation (QVT) Specification, Version 1.3*. OMG Document Number formal/2016-06-03 (https://www.omg.org/spec/QVT/1.3). 2016.

[Obj16c] Object Management Group (OMG). *Object Constraint Language (OCL), Version 2.4*. OMG Document Number formal/2014-02-03 (https://www.omg.org/spec/OCL/2.4). 2016.

[Obj16d] Object Management Group (OMG). *Unified Modeling Language (UML), Version 2.5.1*. OMG Document Number formal/2017-12-05 (https://www.omg.org/spec/UML/2.5.1). 2016.

[Pat69] Keith Paton. "An algorithm for finding a fundamental set of cycles of a graph". In: *Communications of the ACM* 12.9 (1969), pp. 514–518.

[Po01] John D Poole. "Model-driven architecture: Vision, standards and emerging technologies". In: *Workshop on Metamodeling and Adaptive Object Models, ECOOP*. Vol. 50. Citeseer. 2001.

[Ren15] Andreas Rentschler. "Model Transformation Languages with Modular Information Hiding". PhD thesis. 2015. 360 pp. ISBN: 978-3-7315-0346-0. DOI: 10.5445/KSP/1000045910.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[Rut10] Adrian Rutle. “Diagram predicate framework: A formal approach to MDE”. In: 2010.

[Sch94] Andy Schürr. “Specification of graph translators with triple graph grammars”. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 1994, pp. 151–163.

[Sel03] Bran Selic. “The pragmatics of model-driven development”. In: *IEEE software* 20.5 (2003), pp. 19–25.

[SK03] Shane Sendall and Wojtek Kozaczynski. “Model transformation: The heart and soul of model-driven software development”. In: *IEEE software* 20.5 (2003), pp. 42–45.

[Sin+13] Harpreet Singh et al. “Real-life applications of fuzzy logic”. In: *Advances in Fuzzy Systems* 2013 (2013).

[Smu12] Raymond R Smullyan. *First-order logic*. Vol. 43. Springer Science & Business Media, 2012.

[SZ01] George Spanoudakis and Andrea Zisman. “Inconsistency management in software engineering: Survey and open research issues”. In: *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*. World Scientific, 2001, pp. 329–380.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[Ste+08] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[Ste08] Perdita Stevens. “A Landscape of Bidirectional Model Transformations”. In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 408–424. ISBN: 978-3-540-88643-3. DOI: 10.1007/978-3-540-88643-3_10.

[Ste10] Perdita Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software & Systems Modeling* 9.1 (2010), p. 7.

[Ste17a] Perdita Stevens. “Bidirectional transformations in the large”. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2017, pp. 1–11.

[Ste17b] Perdita Stevens. *Megamodel v0.1 in Bx Examples Repository*. http://bx-community.wikidot.com/examples:megamodel. Date retrieved: 27 Apr 2019. 2017.

[Ste18] Perdita Stevens. “Towards sound, optimal, and flexible building from megamodels”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM. 2018, pp. 301–311.

[Stü+18] Patrick Stünkel et al. “Multimodel correspondence through inter-model constraints”. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. ACM. 2018, pp. 9–17.

[TJ07] Emina Torlak and Daniel Jackson. "Kodkod: A relational model finder". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 632–647.

[TA16] Frank Trollmann and Sahin Albayrak. "Extending model synchronization results from triple graph grammars to multiple models". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2016, pp. 91–106.

[Vea+12] Margus Veanes et al. "Symbolic finite state transducers: Algorithms and applications". In: *ACM SIGPLAN Notices*. Vol. 47. 1. ACM. 2012, pp. 137–150.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. URL: `http://www.voelter.de/data/books/mdsd-en.pdf`.

[Wag+11] Dennis Wagelaar et al. "Towards a general composition semantics for rule-based model transformation". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 623–637.

[WK03] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[Xio+07] Yingfei Xiong et al. "Towards automatic model synchronization from model transformations". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 164–173.

List of Figures

2.1. Principles of model transformations 7
2.2. Kernel of the Ecore meta-metamodel 9
2.3. Dependencies between QVT packages 11
2.4. Example of a consistency network with multiary relations 15

3.1. Two possible representations of a consistency network 22
3.2. Two metamodels, each with one metaclass 23
3.3. Simplified class diagram of QVT-R concepts 31

4.1. Schematic view of the decomposition procedure 35
4.2. Two topologies with an extremal density, adapted from [Kla18] 37
4.3. Independent consistency subgraphs 39
4.4. Removal of a totally redundant consistency relation 41
4.5. Decomposition of a partially redundant consistency relation 42
4.6. Consistency specification for Example 4.3.1 46
4.7. Some possible decompositions of Example 4.3.1 with their sizes 48

5.1. From consistency relation graph to consistency rule 50
5.2. Construction of the metagraph in the decomposition procedure 54
5.3. Use of the metagraph in the decomposition procedure 55
5.4. From QVT-R files to metamodels and transformations 56
5.5. Order of processing of QVT-R relations with invariants 59
5.6. Hierarchical structure of a QVT-R domain 60
5.7. Example of a merge of consistency variables 64
5.8. Two consistency specifications resulting in the same hyperedge 65
5.9. A metagraph of a consistency specification and its dual 72
5.10. Three valid combinations to replace a meta-edge 74
5.11. Cycle basis for the dual of Figure 5.10 75
5.12. Candidate combination (1, 2) in a dual 81

7.1. Classification of resulting specifications 98

List of Tables

3.1. Consistency of a few instances of Figure 3.2 24
3.2. Comparable concepts in QVT-R and consistency definitions 30

4.1. Summary of the influence of the decomposition procedure 38
4.2. Summary of decomposition methods 43

5.1. Similar concepts in constraint satisfaction and consistency preservation . 52

6.1. Correspondence between primitive datatypes in OCL, Ecore and Z3 . . . 86

7.1. Example scenarios considered in the evaluation of applicability 101

Listings

3.1. Consistency specification of Example 3.1.1 with QVT-R 26
3.2. Extension of Listing 3.1 with a `where` clause 28
3.3. Three domain patterns 32
3.4. Using QVT-R variables to model consistency rules 33

5.1. A ternary consistency rule with QVT-R 51
5.2. Processing of domain patterns for Example 5.3.3 68

6.1. Recursive construction of an OCL sequence using a Z3 array 87
6.2. Recursive construction of an OCL sequence using a Z3 algebraic datatype 88

List of Algorithms

5.1. Merge of consistency variables 63
5.2. Enumeration of combinations of meta-edges 76

A. Appendix: Translation of OCL Operations

A.1. Arithmetic Operations

Addition (for two number expressions `e1` and `e2`)	
OCL	`e1 + e2`
Z3	`(+ e1 e2)`

Substraction (for two number expressions `e1` and `e2`)	
OCL	`e1 - e2`
Z3	`(- e1 e2)`

Multiplication (for two number expressions `e1` and `e2`)	
OCL	`e1 * e2`
Z3	`(* e1 e2)`

Euclidean division (for two integer expressions `e1` and `e2`)	
OCL	`e1.div(e2)`
Z3	`(/ e1 e2)`

Division (for two number expressions `e1` and `e2`)	
OCL	`e1 / e2`
Z3	`(/ e1 e2)`

Modulo (for two integer expressions `e1` and `e2`)	
OCL	`e1.mod(e2)`
Z3	`(mod e1 e2)`

Absolute value (for a number expression `e1`)	
OCL	`e1.abs()`
Z3	`(ite (< e1 0) (- e1) e1)`

A.2. Boolean Operations

Logical negation (for a boolean expression `e1`)	
OCL	`not e1`
Z3	`(not e1)`

Logical conjunction (for two boolean expressions `e1` and `e2`)	
OCL	`e1 and e2`
Z3	`(and e1 e2)`

Logical disjunction (for two boolean expressions `e1` and `e2`)	
OCL	`e1 or e2`
Z3	`(or e1 e2)`

Logical exclusive disjunction (for two boolean expressions `e1` and `e2`)	
OCL	`e1 xor e2`
Z3	`(xor e1 e2)`

Logical implication (for two boolean expressions `e1` and `e2`)	
OCL	`e1 implies e2`
Z3	`(=> e1 e2)`

A.3. Conversion Operations

Floor function (for a real expression `e1`)	
OCL	`e1.floor()`
Z3	`(to_int e1)`

Round function (for a real expression `e1`)	
OCL	`e1.round()`
Z3	`(to_int (+ 0.5 e1))`

A.4. Equality Operators

Equal-to operator (for any two expressions `e1` and `e2`)	
OCL	`e1 = e2`
Z3	`(= e1 e2)`

Not-equal-to operator (for any two expressions `e1` and `e2`)

OCL `e1 <> e2`

Z3 `(not (= e1 e2))`

A.5. Order Relations and Extrema

Less than operator (for two number expressions `e1` and `e2`)

OCL `e1 < e2`

Z3 `(< e1 e2)`

Less than or equal to operator (for two number expressions `e1` and `e2`)

OCL `e1 <= e2`

Z3 `(<= e1 e2)`

Greater than operator (for two number expressions `e1` and `e2`)

OCL `e1 > e2`

Z3 `(> e1 e2)`

Greater than or equal to operator (for two number expressions `e1` and `e2`)

OCL `e1 >= e2`

Z3 `(>= e1 e2)`

Minimum function (for two number expressions `e1` and `e2`)

OCL `e1.min(e2)`

Z3 `(ite (< e1 e2) e1 e2)`

Maximum function (for two number expressions `e1` and `e2`)

OCL `e1.max(e2)`

Z3 `(ite (> e1 e2) e1 e2)`

A.6. Collection Operations

A.6.1. Operations For Collections

Is-empty function (for a set or a sequence `s`)

OCL `s->isEmpty()`

Z3 `(= 0 (length s))`

Is-not-empty function (for a set or a sequence `s`)	
OCL	`s->notEmpty()`
Z3	`(not (= 0 (length s)))`

A.6.2. Operations For Sequences

Retrieve the first element (for a sequence `s`)	
OCL	`s->first()`
Z3	`(select (array s) 0)`

Retrieve the last element (for a sequence `s`)	
OCL	`s->last()`
Z3	`(select (array s) (- (length s) 1))`

Retrieve the n^{th} element (for a sequence `s` and an integer expression n)	
OCL	`s->at(n)`
Z3	`(select (array s) n)`

A.6.3. Operations For Sets

Union (for two sets `s1` and `s2`)	
OCL	`s1->union(s2)`
Z3	`(mkSet (+ (length s1) (length s2)) (union (array s1) (array s2)))`

Intersection (for two sets `s1` and `s2`)	
OCL	`s1->intersection(s2)`
Z3	`(mkSet (ite (> (length s1) (length s2)) (length s1) (length s2)) (intersect (array s1) (array s2 )))`

Difference (for two sets `s1` and `s2`)	
OCL	`s1->excludesAll(s2)`
Z3	`(mkSet (length s1) (difference (array s1) (array s2)))`

Symmetric difference (for two sets `s1` and `s2`)

OCL
```
s1->symmetricDifference(s2)
```

Z3
```
(mkSet (+ (length s1) (length s2)) (difference
(union (array s1) (array s2))
(intersect (array s1) (array s2 ))))
```

A.7. String Operations

Concatenation (for two strings `s1` and `s2`)

OCL `s1 + s2`, `s1.concat(s2)`

Z3 `(str.++ s1 s2)`

Substring (for a string `s1` and two indexes, `l` (lower) and `u` (upper, included))

OCL `s1.substring(l, u)`

Z3 `(str.substr s1 l (+ (- u l) 1))`

Length (for a string `s1`)

OCL `s1.size()`

Z3 `(str.len s1)`

Conversion to integer (for a string `s1`)

OCL `s1.toInteger()`

Z3 `(str.to.int s1)`