Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH Dissertation No. 17643

Using Patterns to Develop Consistent Design Constraints

Michael Wahler, Dipl.-Inf.
Department of Computer Science

2008

DISS. ETH NO. 17643

USING PATTERNS TO DEVELOP CONSISTENT DESIGN CONSTRAINTS

A dissertation submitted to

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of

Doctor of Sciences

presented by

MICHAEL WAHLER

Dipl.-Inf., TU München

born on March 10th, 1978

German citizen

accepted on the recommendation of

Prof. Dr. David Basin, examiner

Prof. Dr. Peter Müller, co-examiner

Dr. Jana Koehler, co-examiner

2008

Acknowledgements

Yksi tyhmä kysyy enemmän
kuin kymmenen viisasta ehtii vastata.

Finnish proverb

One stupid person asks more than ten wise ones can spare time answering. I want to thank those who supported me in this venture with their wisdom, time, and patience.

I thank my academic supervisor, Prof. David Basin, and my industrial mentor, Dr Jana Koehler, who gave me the chance to pursue a PhD and actively supported me from my first steps on in the world of research. Above all, Prof. Basin showed me how to render initial ideas into concise, detailed contributions and Dr Koehler taught me how to properly present research in written and spoken form.

I thank Prof. Tobias Nipkow for his inspiring courses and seminars, which contributed to my decision of pursuing a PhD and taking a closer look at formal languages.

I thank Dr Achim Brucker for his support in formalizing things in HOL-OCL. In addition, Achim's LaTeX wisdom substantially helped me improve the layout of this thesis.

I thank my colleagues from the InfSec group at ETH Zurich for providing such an excellent professional and personal atmosphere. In particular, I thank Dr Paul Hankes Drielsma for insizzles into North American culture, Dr Manuel Hilty for insights into Swiss culture, Dr Boris Köpf for 865 research meetings, Dr Alexander Pretschner for his advice on setting priorities, and Dr Paul Sevinç for introducing me to Zurich's largest Schnitzels.

To the same extent, I thank my colleagues at the IBM Zurich Research Laboratory who contribute to making the Lab a unique place to work. In particular, I thank Christian Gerth for sharing the model and constraints that I used for the case study about the process-merging constraints, Dr Jochen Küster for advising me in strategic matters, Ulrich Schimpel for stimulating my creativity, Jussi Vanhatalo for his entertainment with the driest jokes ever, and Olaf Zimmermann for rocking and rolling with me.

I thank my parents for helping me polish the German translation of the abstract of this thesis, but most of all, for their unlimited support and unconditional love. Life is a lot easier with a backup of this magnitude.

Ksenia, I thank your for all your emotional and scientific support. You let the sun shine for me every day.

Contents

Abstract

Developing constraint specifications for class models is a time-consuming and error-prone task because typical specifications contain numerous constraints, which in addition often state complex facts about the elements of the model. As the size and the complexity of constraint specifications grow, so does the probability of inadvertently specifying inconsistent models that cannot be instantiated because of contradictory constraints.

In this thesis, we introduce a novel approach to developing consistent constraint specifications based on constraint patterns. The input for this approach is an unconstrained class model and the output is a formal constraint specification. The approach comprises four phases. First, class models are automatically analyzed to elicit potentially missing constraints. Second, a library of composable constraint patterns allows developers to write concise constraint specifications based on the results from the elicitation phase. Third, this approach contains consistency assertions on the constraint pattern library that enable automatic consistency analysis of pattern-based constraint specifications. As basis for these consistency observations, we provide formal definitions of consistency properties of constrained class models. Fourth, pattern-based constraint specifications are transformed into logical expressions or code in a programming language.

The focus of the approach is on effectiveness and practicability. Therefore, we introduce a tool that allows model developers to follow the theoretic approach in a guided way and effectively apply it in modeling projects. We use this tool to conduct several case studies in which we validate that our approach improves the state-of-the-art constraint development in terms of detecting missing constraints, shortening development time, avoiding inconsistencies, and yielding more comprehensible constraints.

Zusammenfassung

Die Entwicklung von Constraintspezifikationen für Klassenmodelle ist zeit raubend und fehleranfällig, weil typische Spezifikationen zahlreiche Constraints enthalten, die darüber hinaus oft komplizierte Eigenschaften von Modellelementen beschreiben. Mit steigender Grösse und Komplexität von Constraintspezifikationen steigt auch die Wahrscheinlichkeit, versehentlich inkonsistente Modelle zu spezifizieren, die auf Grund widersprüchlicher Constraints nicht instantiiert werden können.

In dieser Dissertation führen wir einen neuartigen Ansatz zur Entwicklungen von konsistenten Constraintspezifikationen ein, der auf Constraintmustern basiert. Die Eingabe für diesen Ansatz ist ein constraintfreies Klassenmodell, und die Ausgabe ist eine formelle Constraintspezifikation. Dieser Ansatz beinhaltet vier Phasen: 1. Klassenmodelle werden automatisch analysiert, um potentiell fehlende Constraints zu eruieren. 2. Eine Bibliothek von zusammensetzbaren Constraintmustern erlaubt es Entwicklern, auf der Basis dieser Eruierung knappe und präzise Constraintspezifikationen zu entwerfen. 3. Dieser Ansatz beinhaltet Konsistenzaussagen über diese Bibliothek von Constraintmustern, die eine automatische Konsistenzanalyse von musterbasierten Constraintspezifikationen ermöglichen. Als Grundlage für diese Konsistenzbetrachtungen erstellen wir formelle Definitionen von Konsistenzeigenschaften constraint-annotierter Klassenmodelle. 4. Musterbasierte Constraintspezifikationen werden in logische Ausdrücke oder Code in einer Programmiersprache übersetzt.

Der Schwerpunkt dieses Ansatzes liegt auf Anwendbarkeit. Deshalb stellen wir ein Werkzeug vor, das Modellentwickler bei der Benutzung des theoretischen Ansatzes führt, um ihn effektiv in Modellierungsprojekten einzusetzen. Wir benutzen dieses Werkzeug zur Durchführung mehrerer Fallstudien, in denen wir validieren, dass unser Ansatz den heutigen Stand der Constraintspezifizierung verbessert, indem er beim Finden von fehlenden Constraints hilft, die Entwicklungszeit verkürzt, Inkonsistenzen vermieden werden und leichter verständliche Constraints erstellt werden können.

Chapter 1

Introduction

1.1 Problem Statement

Writing software is a difficult and complex task. In order to simplify this task, various programming paradigms and languages have been developed. Whereas early computers needed to be programmed in binary machine code, assembly languages [Saxon and Plette, 1962], also called second-generation programming languages, were developed to provide a more abstract interface to computer programming. Although assembly code is typically better readable than machine code, it remains unstructured and hard to understand. Modern programming paradigms such as functional programming [Bird and Wadler, 1988] or object-oriented programming [Cox, 1986], also called third-generation programming languages, abstract from low-level concepts such as CPU instructions and memory-address calculations by providing specialized syntax for various programming tasks. Examples for third-generation languages are Fortran, C, and Java. With increasing levels of abstraction, programmers can focus on the domain of a problem and on finding a solution for it, agnostic of technical details. This is one of the motivations behind the development of fourth-generation programming languages [Martin, 1982] such as SQL or PostScript, which focus on specific application domains.

A recent development approach that takes these concepts to another level of abstraction is Model-Driven Engineering (MDE). In MDE, certain aspects of a system, e. g., security requirements [Basin et al., 2006], are initially specified in terms of graphical models at a high level of abstraction. During the development process, these models are incrementally refined and eventually transformed into code in some programming language, which can be complemented by additional code. This allows developers to abstract from technical details at the beginning of the development process, which has two advantages. First, it makes the specification more accessible for domain experts without technical background. Second, by abstracting from technical details, the developer can focus on the data structures and algorithms necessary for the solution of a problem. In addition, with MDE, the complexity of a system can be dealt with by using different model types, which can be chosen accordingly for different aspects of the system.

Despite these improvements, the system under development remains complex, and most graphical languages reach their limits in terms of expressiveness when it comes to modeling complex system behavior or structures. To this end, graphical languages can be complemented with textual constraint languages, which are used to annotate graphical elements with textual expressions. This allows model developers to express details of the system not expressible with graphical languages. There are two kinds of constraints:

Domain-independent constraints occur in models of any domain and are required because of limitations of diagrammatic languages. For example, restricting the number of objects in a relation to an attribute value can typically not be expressed in terms of diagrammatic modeling languages. *Domain-specific* constraints originate in their respective domain and are usually valid in their own domains only. There are different kinds of domain-specific constraints, for example, legal restrictions that a system needs to obey, company policies that grant privileges to certain kinds of customers, technical restrictions on a system, e.g., features not yet implemented, or security restrictions.

Model developers must be experts in the application domain in order to understand the semantics of each constraint and at the same time, they must be experts in object-oriented modeling and know how to map the domain semantics of the constraints onto the modeling and constraint languages used. Due to these difficulties, developing constraint specifications is a time-consuming and error-prone task. Furthermore, writing textual constraints can be considered an atavism from traditional code-based development, and whereas system properties such as the structure or the object life cycle can be represented in terms of diagrammatic languages, constraints must be specified as textual expressions.

Due to the quantity and intricacy of the system elements to be constrained, constraint specifications can contain constraints that are inadvertently contradictory, which makes the specification inconsistent. Such inconsistencies may only be detected when the code generated from the model is tested, which requires costly development iterations. Besides this, the role of consistency in the MDE process is vague because of three reasons. First, there is only a restrictive definition of consistency of constraint specifications in the literature. Second, automatic consistency analyses are incomplete because practically relevant constraint languages are undecidable. Third, there is no definition for a process of consistent model refinement.

In this thesis, we introduce a model-driven approach to developing constraint specifications for class models based on *constraint patterns*. Using constraint patterns, our approach accelerates the development and maintenance of constraint specifications because it abstracts from concrete textual syntax and thus reduces typical syntactic and semantic errors by providing predefined constraint templates. Furthermore, it enables a systematic approach to model refinement, i. e., complementing graphical models with textual constraints, because recurring problems can be automatically detected and patterns can be suggested for partial remediation.

Since constraint specifications comprise logical expressions, they can potentially contain contradictory constraints. Therefore, we investigate different notions of consistency for constraint specifications and show how constraint patterns enable model developers to refine models in a consistent way. We provide tool support for our approach in the form of an extension for a model-driven development tool that supports users in the process of consistent model refinement. Finally, we perform case studies in which we apply our approach to real-world models.

Our approach is geared to domain specialists, e. g., business analysts, who have an understanding of graphical modeling languages, but no expertise in logics or declarative programming languages. With our approach, we aim at providing a means for domain specialists to develop concise and consistent constraint specifications without the need to acquire formal specification languages.

1.2 Contributions

The overall contribution made in this thesis is a novel MDE process that provides a method and tools for the development of concise and consistent constraint specifications. This process supports model developers through four phases, which we illustrate in Figure 1.1 and further explain in the following.

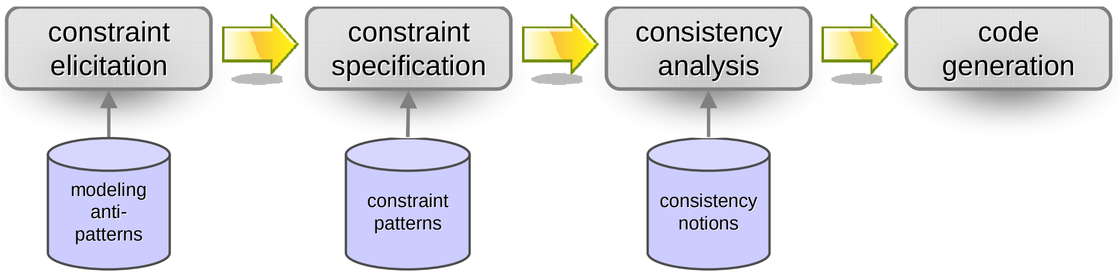


Figure 1.1: Approach for developing concise and consistent specifications.

Constraint Elicitation.

In the constraint elicitation phase, graphical models are examined for missing textual constraints. Our contribution for this phase is a method that supports model developers in finding potentially missing constraints in class models through an automatic analysis. This contribution improves on the state of the art by

- showing how to identify limitations on the expressiveness of graphical modeling languages and capture them as anti-patterns,
- introducing a set of anti-patterns that frequently occur in unconstrained class models, and
- explaining how these anti-patterns can be remedied by enriching class models with textual constraints.

Constraint Specification.

After the missing constraints have been identified, the model developer must specify these constraints in a formal language such that model instances can be automatically evaluated against the constraints. Our contribution for the constraint specification phase is the concept of composable constraint patterns, which allow model developers to specify constraints by concise graphical means. This contribution improves on the state of the art by

- extending the concept of constraint patterns to *composable* constraint patterns, which provide a substantially higher expressiveness than existing constraint-pattern approaches,
- showing how the semantics of constraint patterns can be precisely defined in terms of functions in higher-order logics, and
- providing an extensible library of constraint patterns that cover typical specification tasks, in particular remedying the anti-patterns used for constraint elicitation.

Consistency Analysis.

After the constraint specification has been completed, it must pass a *consistency analysis*, which detects inconsistencies such as contradictions. Our contribution is a novel approach to automatic consistency analysis of pattern-based constraint specifications. This contribution improves on the state of the art by

- identifying and formalizing distinct precise definitions for consistency in the context of constrained class models,
- introducing the concept of consistency theorems for constraint patterns, which are the basis for an automatic, heuristic analysis, and
- developing a heuristic approach for automatically analyzing the consistency of pattern-based constraint specifications in polynomial time. This approach is based on the consistency theorems for the given library of constraint patterns.

Code Generation and Tool Support.

In the fourth and last phase of our method, *code generation*, pattern-based constraint specifications are transformed into statements in a textual specification language. Our contribution is an automatic model transformation and an extension to the development tool IBM Rational Software Architect (RSA) that allow model developers to effectively use the development approach introduced in this thesis. This contribution improves on the state of the art by

- introducing a constraint elicitation component that analyzes class models for occurrences of the previously defined anti-patterns and presents the results in a user-friendly way,
- presenting an implementation of our constraint-pattern library that allows model developers to use constraint patterns as graphical model elements,
- developing an "instant fix" mechanism based on constraint patterns that allows model developers to remedy occurrences of anti-patterns by a single action,
- presenting an implementation of the heuristic consistency analysis that displays inconsistencies in a user-friendly way, and
- implementing a transformation that generates textual constraints from pattern-based constraint specifications.

1.3 Outline

In the following, we describe the organization of this thesis and briefly summarize each chapter.

Chapter 2: Background and Related Work.

In this chapter, we provide background information on Model-Driven Engineering (MDE), and in particular, on standard modeling languages such as the Unified Modeling Language

(UML) and the Meta Object Facility (MOF). We give an introduction to the Object Constraint Language (OCL), the standard constraint language for UML and MOF. Furthermore, we introduce patterns in software development, gives an overview of consistency in classical logics, and briefly introduce interactive theorem proving.

Chapter 3: Constraint Elicitation and Specification.

In this chapter, we motivate the need for model refinement. Since graphical modeling languages do not offer sufficient expressiveness for certain details, graphical models must be annotated with textual constraints. We illustrate typical problems of graphical class-modeling languages and derive anti-patterns from them. Subsequently, we show how class models can be augmented with textual constraints to remedy these anti-patterns.

Chapter 4: Specifying Constraints Using Patterns.

In this chapter, we present the concept of an extensible library of composable constraint patterns. Besides *elementary* constraint patterns as known in the literature, such a library contains *composite* constraint patterns. These can be used to combine pattern instances and thus create complex constraints. We show how the semantics of constraint patterns can be defined in higher-order logic and present an example library of constraint patterns that we consider useful in practice.

Chapter 5: Consistency of Constraint Specifications.

In this chapter, we investigate the notion of consistency of constraint specifications. As it turns out, the classical notion of consistency has limited relevance for object-oriented constraint specifications and therefore, we develop fine-grained notions of consistency for object-oriented constraint specifications. Furthermore, we show how the consistency of a given constraint specification can be analyzed by interactive theorem proving and how the degree of proof automation can be increased by using previously proven lemmas about the constraint patterns.

Chapter 6: Consistent Model Refinement Using Patterns.

In this chapter, we examine how the consistency of constraint specifications can be established in the development process. We first provide an overview of different analysis methods and explain their advantages and disadvantages. Subsequently, we describe how constraint patterns can be used to refine class models in a consistent way. To this end, we analyze the dependencies between the constraint patterns introduced in Chapter 4 to detect potential inconsistencies. Furthermore, we show how such refinement integrates into the MDE process.

Chapter 7: Tool Support.

In this chapter, we present tool support for our approach that integrates into the MDE tool IBM Rational Software Architect (RSA). The tool support comprises constraint elicitation to detect elements that potentially require refinement, a library of constraint patterns, consistency analysis for instances of the constraint patterns, and code generation that transforms a pattern-based constraint specification into a standard specification language.

Chapter 8: Validation.

In this chapter, we perform three case studies in which we refine class models by developing constraint specifications. For carrying out these case studies, we follow the approach introduced in this thesis and use the tools developed. Subsequently, we evaluate our approach against quantitative and qualitative criteria and report on general insights.

Chapter 9: Conclusion.

In this chapter, we summarize the achievements made in this thesis, the limitations of our approach, and report on the lessons learned in the course of developing the approach. We furthermore point out possibilities where future work can improve or extend the approach developed in this thesis.

Chapter 2

Background and Related Work

In this chapter, we present background information for this thesis and discuss related work for the respective topics. Further related work is discussed in the remaining chapters of this thesis where appropriate.

In Section 2.1, we present the concept of Model-Driven Engineering (MDE) in which systems are specified by a set of models from which code is automatically generated. We define the notion of a model as used in this thesis in Section 2.2. In particular, we introduce class models as used in the Meta Object Facility (MOF) and the Unified Modeling Language (UML). In MDE, class models often need to be augmented by textual constraints. Therefore, we give an overview of the Object Constraint Language (OCL), a textual constraint language that is typically used to annotate class models in Section 2.3.

Patterns have become an important means of specification in MDE and other domains. In Section 2.4, we present the concept of patterns in general, and the concept of constraint patterns, which can be used for specifying constraints on models, in particular.

Models and their constraint specifications must be consistent. In Section 2.5, we therefore investigate the notion of consistency in classical logics and in MDE. Since practicable constraint languages are typically undecidable, interactive theorem proving provides means for formally carrying our proofs. In Section 2.6, we explain the concept of interactive theorem proving in general and of HOL-OCL, a proof environment for OCL specifications, in particular.

2.1 Model-Driven Engineering (MDE)

With the term Model-Driven Engineering (MDE), we denote software development approaches in which the system to be developed is specified in terms of (graphical) models. MDE development processes closely follow traditional software engineering processes, but use models as first-class artifacts compared to code in traditional processes. Our notion of MDE thus embraces concepts such as Model-Driven Architecture (MDA) [Kleppe et al., 2003], Model-driven development (MDD), and the "original" concept of MDE [Kent, 2002].

In Figure 2.1, we show an abstract view of a typical software engineering process. Such a process starts with a *requirements engineering* phase in which the functional and non-functional requirements of the system under development are elicited. In the *analysis* phase, structural and functional models of the system are built. This task comprises identifying objects and specifying control flow, for example. In the *design* phase, these models are refined, for example, by identifying subsystems of the system or implementing algorithms. Eventually, the system is *tested* and *deployed*, which starts a maintenance cycle. Maintaining software can lead back to adapting the requirements and accordingly go through the development process again.

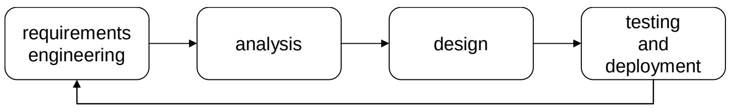


Figure 2.1: Abstract view of a typical software engineering process.

MDE processes extend software engineering processes as follows. In the analysis phase, initial models are created, typically at a high level of abstraction with a graphical syntax. In the design phase, these abstract models are made more concrete. This can be achieved by adding further model elements or annotating models with textual constraints. Thus, the level of *abstraction* of the models decreases in the course of the MDE process, whereas the level of *maturity*, which is defined as the reciprocal of the abstraction level in [Kleppe and Warmer, 2003], increases accordingly. We define these concepts *informally* as follows and provide a formal definition in Section 2.3.

Definition 1 (Abstraction / Refinement). *Models are refined by adding details in the form of further models elements, which decreases their level of abstraction and increases their level of maturity to the same extent.*

Once models have reached a satisfactory level of maturity, they are automatically transformed into code, which can subsequently be tested and deployed. In Figure 2.2, we illustrate these concepts. Going top down, different kinds of models are specified at a high level of abstraction and subsequently refined by a sequence of refinement steps, indicated by *refinement**. Eventually, the refined models are transformed into code, which is also called *code generation*.

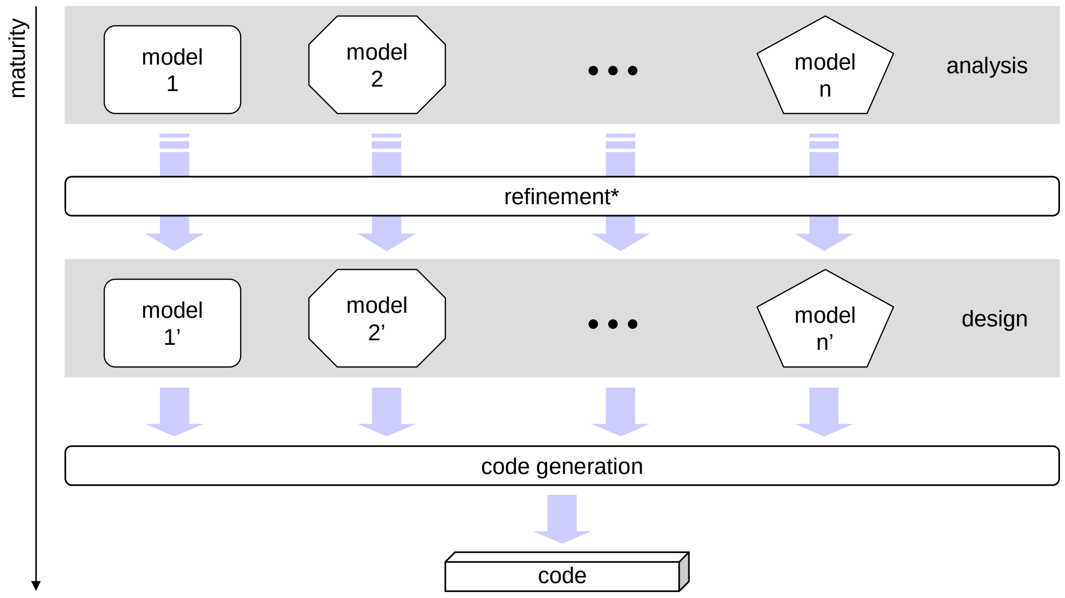


Figure 2.2: Refinement and code generation in MDE.

In the remainder of this chapter, we provide background information on the concepts contained in MDE. We start by defining what a model is.

2.2 Models and Meta-Models

Narrowing the definition in [Rumbaugh et al., 1991], we define the term *model* as follows.

Definition 2 (Model). *A* model *is a formal abstraction of a system.*

In software engineering, models are used to describe the structural and behavioral aspects of a system. The Unified Modeling Language (UML) [Object Management Group (OMG), 2006c] provides a family of model types to model these different aspects. For example, it comprises class models for the structural aspects and activity diagrams for the behavioral aspects for a system. In UML, two types of models can be augmented with textual constraints: class models and state machines. In this thesis, we focus on class models, which we define as follows.

Definition 3 (Class Model). *A class model represents the concepts of a system in terms of classes and associations between the classes.*

In Figure 2.3, we present the MOF meta-model as an example class model. We assume that the reader is familiar with the basic concepts of class modeling, but we introduce the most important concepts here. Class models comprise *classes* such as NamedElement or Type, which are graphically represented by rectangles. In these rectangles, the *attributes* and *operations* of the respective class are shown. For example, NamedElement has one attribute called name of type String. Classes can be related by generalization relations, which express subtype relations between classes. These relations are denoted by a white triangle. For example, Type is a *subtype* (or *subclass*) of NamedElement. Classes can also be related by different kinds of relations, which can be directed such as the superClass relation or they can denote composition, indicated by a black diamond such as the ownedAttribute association. We will introduce the remaining concepts where needed.

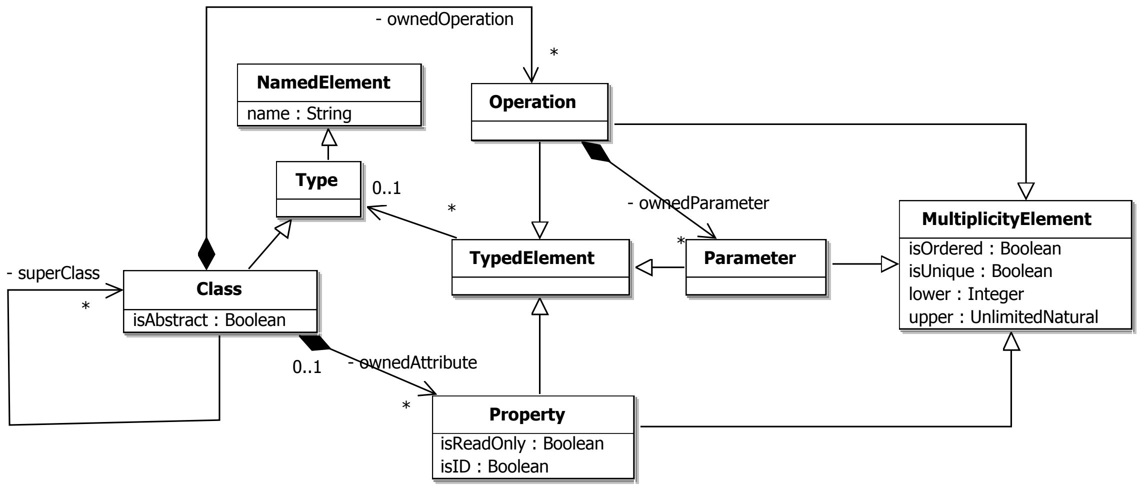


Figure 2.3: Extract of the Essential MOF (EMOF) meta-model.

The possible elements of a model and their relations are defined in a *meta-model*. The Meta Object Facility (MOF) [Object Management Group (OMG), 2006a] is a standard that defines the building blocks of (meta-)modeling. Its core, the EMOF, defines the facilities that are commonly found in object-oriented approaches such as types, classes, properties, and operations. Meta-models, including EMOF, are typically defined in terms of class models.

MOF defines a hierarchy of model abstractions, which can comprise up to four layers [Atkinson and Kühne, 2003]. In general, a model in layer n is called an *instance* of the model in layer $n + 1$, which in turn is called its *meta-model*. In Figure 2.4, we illustrate these four modeling layers.

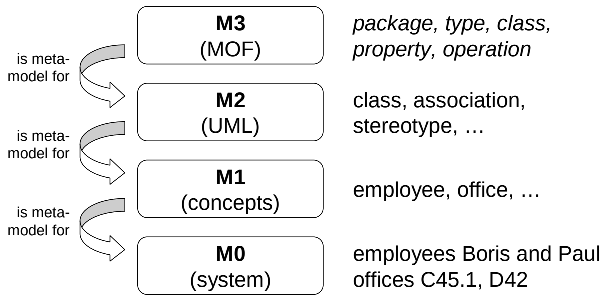


Figure 2.4: The four modeling layers of MOF.

The most abstract layer, commonly perceived as *M3*, is the MOF meta-model, as shown in Figure 2.3. It defines the core modeling concepts and is defined recursively, i. e., a model on this layer is an instance of itself [Seidewitz, 2003].

MOF is considered a meta-meta-modeling language, i. e., it is commonly used to define meta-models. A prominent example of an *M2* layer meta-model is the UML family of meta-models. It comprises meta-models such as the class model, which defines modeling elements such as n-ary associations or stereotypes [Object Management Group (OMG), 2006c].

The models in layer *M1* define the *concepts* of a system. For example, the structural concepts of a company can be modeled in terms of a class model and comprise classes such as Employee or Office.

The most concrete layer *M0* represents the elements of some concrete *system*. For example, the model of a company could comprise elements such as an employee called "Boris" and an office labeled "C45.1". *M0* models are instances of *M1* models.

The most important model type in this thesis is the class model. As can be seen in Figure 2.3, a class comprises a set of attributes (modeled as Property) and a set of operations. A class is a special Type. MOF and UML distinguish between class types and primitive types such as String or Boolean. Relations between concepts are modeled as n-ary associations between classes in UML, whereas MOF does not provide means for defining n-ary associations. In this thesis, we cover binary associations only, which can be modeled in terms of MOF as follows. A binary association between classes A and B is modeled by an attribute b that is owned by A and an attribute a owned by B. The cardinality of such a relation is expressed using the attributes lower and upper bound that Property inherits from MultiplicityElement.

2.2.1 Example Model.

In this subsection, we present a class model that we will use as example throughout the remainder of this thesis. In Figure 2.5, we illustrate a model company using the concrete syntax of UML class diagrams.

In detail, the elements are defined as follows. Employee, Manager, Office, Single and Cubicle are instances of the MOF concept *class*. Attributes of classes are represented as properties of this class. In particular, name, salary, budget, headCount, isCEO, desks are MOF *properties* in the company model. Properties are also used for defining associations

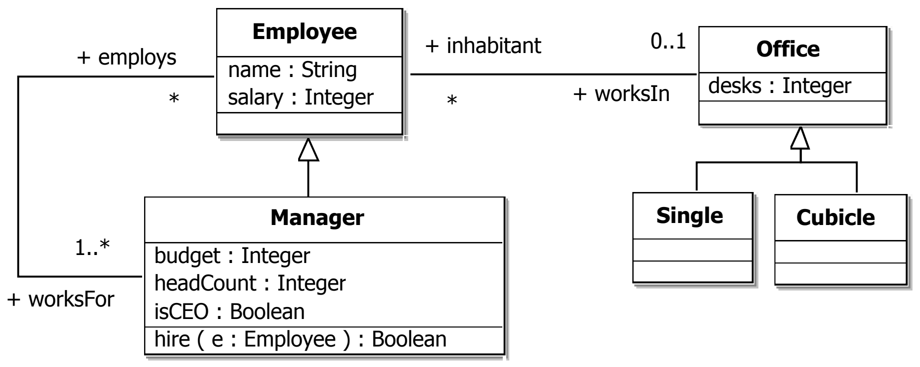


Figure 2.5: Model "company": example instance of MOF.

between classes, which reflect relations between objects. In particular, employs, worksFor, inhabitant, and worksIn represent such properties, which are also called *association ends*.

String, Integer and Boolean are primitive *types* provided by the MOF meta-model. hire is a MOF *operation* whose return type is Boolean. Operations can have parameters. An example is parameter e for the hire operation. 1..$*$, 0..1 and $*$ are MOF *multiplicity elements*. Properties, operations, and parameters are special kinds of multiplicity elements, i. e., their values can be sets.

The generalization relations between Manager and Employee and Single/Cubicle and Office are instances of the MOF *superClass* relation. This relation is used to model subtype relationships between classes.

As shown, the elements in this model are instances of MOF elements, i. e., we use the MOF as meta-model for the company model. The company model can also be expressed in terms of the UML meta-model for class models. In the remainder of this thesis, we refer to the term *class model* as that type of model that can be expressed in terms of the UML meta-model for class models, which subsumes the MOF meta-model.

2.2.2 Instances of Class Models.

Each instance of a class model comprises a set of *objects*, which denote phenomena in a system, their attribute values, and a set of *links*, which relate two objects and are instances of a binary association between the classes of these objects. To avoid confusion about the overloaded term *instance*, we refer to an instance of a class model as a *state* of the class model in the remainder of this thesis and define it as follows.

Definition 4 (State). *The state $\tau : M$ of a class model M is defined by a set of objects and links between these objects. The type of each object $o \in \tau$ is defined as a class $C \in M$ and the type of each link $l \in \tau$ is defined as an association $A \in M$. A state can be represented as a directed graph in which the objects of the state represent the nodes and the links represent the edges.*

We present an example state τ for the company model in Figure 2.6 in the form of an *object diagram*. This state comprises four objects (m_1, e_1, s_1, c_1) and three links $\big((m_1, e_1), (m_1, s_1), (e_1, s_1)\big)$. The types of the objects are indicated behind the object identifier separated by a colon. The types of the links are implied by the types of the objects. Thus, $(m_1, e_1) \in$ Manager $\times$ Employee , $(m_1, s_1), (e_1, s_1) \in$ Employee $\times$ Office .

Where appropriate, we use a more mathematical notation for the previously introduced concepts. If m_1 denotes an object of class Manager, $employs(m_1)$ represents the

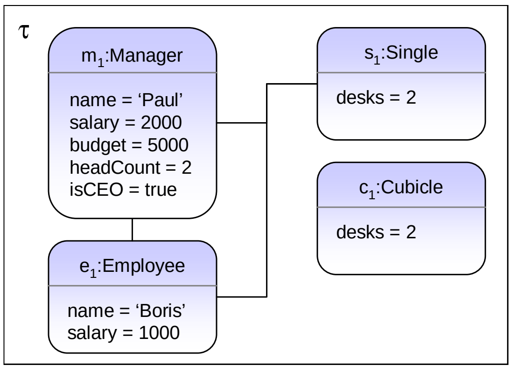


Figure 2.6: Example state of the company model.

set of all objects of type Employee to which m_1 is related. We further define a function $\_^{-1}$ that denotes the opposite end of an association. For example, $employs^{-1}$ denotes the association end worksFor.

In order to rule out undesired objects in model states, models can be augmented with textual constraints. In the following section, we give an overview of OCL, the standard constraint language in the context of MDE.

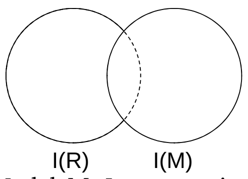
2.3 The Object Constraint Language (OCL)

While models were solely used for documentation and communication purposes in the beginning of MDE, recent model-centric development approaches use models as first-class artifacts in the development process. For example, business process models can be transformed to executable code that is run on process execution engines [Hauser and Koehler, 2004] or models in a domain-specific security language are transformed to UML [Basin et al., 2006]. To guarantee correctness of the execution of the generated code, it is crucial that every model state conforms to its defining model, which should represent the concepts of the underlying system as precisely as possible.

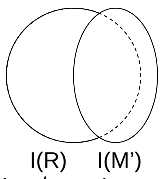
However, modeling languages such as MOF or UML offer only limited support for defining the concepts of a model or a system. Whereas entities and basic relations can be described in terms of types, classes and their properties, relations and dependencies can be further specified by basic multiplicity (i. e., cardinality) constraints only. In order to express complex relations and restrictions in a model, OCL has been introduced [Object Management Group (OMG), 2003] as part of MOF and UML, a textual constraint language for object-oriented modeling languages. OCL is based on a three-valued logic with an explicit element denoting undefinedness and a library providing a typed set theory and basic datatypes. It serves three purposes: First, *invariants* can be specified for classes. An invariant is a predicate that holds for all objects of the constrained class. In this thesis, we use the terms *invariant* and *constraint* synonymously.

Definition 5 (Invariant). *An OCL invariant I is a logical predicate defined on a class C. For each object c of class C, $I(c)$ must hold throughout a sequence of operations, i. e., if $I(c)$ holds before execution an operation o, it must hold after o terminates.*

Adding constraints to class models rules out invalid states, which increases the *maturity level* of the class model [Kleppe and Warmer, 2003, Wahler, 2008]. For giving an intuition for this idea, we use a function I that maps a set of concepts to the set of all possible objects for these concepts. In particular, $I(M)$ denotes the set of all objects in all possible states of a model M and $I(R)$ denotes the set of all possible objects in a real system. Figure 2.7 a) visualizes a model M with a low maturity level: A large part of $I(M)$ is not inside $I(R)$, i. e., $I(M)$ contains many elements that are not representations of the real system. By adding constraints to M, a model M' can be developed with a higher maturity level. Figure 2.7 b) shows that there are significantly less elements in the set $I(M')\setminus I(R)$, which means that less invalid instances can be derived from M' than from M. Thus, M' has a higher maturity level than M.



(a) Model M: Low maturity level.



(b) Model M': High maturity level.

Figure 2.7: Increasing model maturity by specifying constraints.

Definition 6 (Maturity). *A model M' has a higher maturity level than a model M' if the set of all possible objects of states of M' contains less elements than the corresponding set for M, or formally $|I(M')| < |I(M)|$.*

The second purpose of OCL allows model developers to apply the design-by-contract principle [Meyer, 1992] by annotating operations with contracts. Each contract consists of a precondition, which restricts the applicability, and a postcondition, which describes the result of the operation. Whereas pre-conditions specify the conditions that must hold when the execution of an operation is triggered, post-conditions specify the conditions that must hold when the execution of an operation terminates [Object Management Group (OMG), 2006b].

Definition 7 (Contract). *An OCL contract for an operation o of class C comprises two logical predicates pre and $post$ defined on the parameters of o and the properties of C and its superclasses. If pre holds when o is executed, $post$ is guaranteed to hold if o terminates.*

Third, OCL can be used to define additional attributes and operations for class models. Attributes added using OCL are always *derived* attributes, i. e., their values are calculated from the values of other attributes. Operation definitions can be recursive, but they have to be defined such that the defined operator terminates when applied and its pre-condition holds [Object Management Group (OMG), 2003]. Since OCL expressions are always free of side-effects [Object Management Group (OMG), 2003], such user-defined operations are also called *query operations*.

In the following, we illustrate several examples of invariants and contracts for the company model. With these examples, we introduce all language elements of OCL that we use in the remainder of this thesis. First, we define two invariants as follows.

```
context Manager
  inv budgetGreaterZero: self.budget >= 0

context Single
  inv onlyManagers: self.inhabitant->forAll(x | x.oclIsTypeOf(Manager))
```

Invariants are declared by the keyword context followed by the name of the constrained class, the keyword inv, and an optional name of the invariant. The last part of an invariant declaration is a logical statement, in which the keyword self denotes a variable that is universally quantified over all objects of the constrained class.

The first invariant, *budgetGreaterZero* for Manager, states that the budget of managers must not be negative. In a more mathematical syntax, this invariant reads as follows.

$$\forall \text{self:Manager. budget(self)} \geq 0$$

The second invariant, *onlyManagers*, is defined on Single offices. It restricts the inhabitants of these offices to objects of class Manager, which is denoted by an OCL *path expression*. Path expressions, also called *navigations*, consist of a sequence of association ends separated by a dot. In the remainder of this thesis, we use the terms *path* and *navigation* as synonyms for such expressions. Furthermore, we ignore the fact that OCL supports special types of sets such as multi-sets or ordered sets and assume that the type of path expressions is a simple set.

The invariant contains a universal quantification (−>forAll(x | P(x))) over the set of all inhabitants of an office. The elements of this set are bound to the variable x. The expression x.oclIsTypeOf(C) evaluates to true if x is an instance of class C, while x.oclIsKindOf(C) evaluates to true if x is an instance of either C or of a subclass of C.

Contracts specify pre-conditions and post-conditions for operations. An example contract for the operation hire of class Manager reads as follows in OCL.

```
context Manager::hire(e: Employee): Boolean
  pre:  not self.employs−>includes(e)
  post: self.employs = self.employs@pre−>including(e)
```

The precondition of hire requires that the employee who is supposed to be hired is not already employed. This is specified using the boolean operation −>includes(x), which evaluates to true if the set that this operation is invoked on contains x. The precondition further comprises the OCL negation operator not. Alternatively, the precondition could be specified as self.employs−>excludes(e).

The postcondition requires that after the operation has executed, the set of employees is the same as before the execution, except for the new employee who has joined this set. This is accomplished by requiring that the set of employees (self.employs) after the execution of the operation is equal to the set of employees before the execution of the operation, which can be specified using the keyword @pre, except for one new element, e. In OCL, S−>including(e) denotes $S \cup \{e\}$.

We also provide examples for derived attributes and operations, which we define as follows.

```
context Employee
def: isAlone : Boolean = self.worksIn.oclIsTypeOf(Single)

context Manager
def: paysAtLeast(wage:Integer) : Boolean
        = self.employs−>forAll( e | e.salary >= wage )
```

These definitions add a derived attribute isAlone to class Employee. This attribute evaluates to true if the respective employee works in a single office. The operation paysAtLeast for class Manager evaluates to true for managers who pay their employees a salary that is at least as high as the parameter wage.

In the following, we introduce further OCL operations that we will use in the remainder of this thesis. OCL comprises an operator for equality (=) and inequality (<>) as well as the standard arithmetic operators (+, −, >, >=, <, <=).

The size of some set exp is represented by exp−>size() and if the size of exp is zero, exp−>isEmpty() holds, otherwise, exp−>notEmpty() holds. OCL supports typical set operations such as set union (S−>union(S')), intersection (S−>intersect(S')), and set difference (S − S'). If a set S contains numbers from $\mathbb{R}$, S−>sum() represents the sum of all numbers in S. Multi-sets, called *bags* in OCL, and ordered sets can be typecast to simple sets with the operator −>asSet().

The −>collect() operation iterates over collections and aggregates the elements of such collections in another collection. For example, self.employs−>collect(name) invoked on a manager m results in a set of String objects containing the names of all employees of m. Since −>collect() is frequently used, a shorthand notation was introduced [Kleppe and Warmer, 2003], which replaces the collect operation by a dot notation. For example, self.employs−>collect(name) and self.employs.name are equivalent expressions in OCL.

The operator exp−>isUnique(a) holds if a evaluates to a different value for each element in exp. For example, self.employs−>isUnique(name) requires that all employees of some manager have distinct names. If objects need to be uniquely identified by a collection of attributes, tuples can be used as parameters for isUnique. For example, self.employs−>isUnique(Tuple(x=name,y=salary)) holds if a manager does not have two employees with the same name and the same salary.

Analogous to the universal quantifier −>forAll(), OCL provides an operator for existential quantification over a set of elements. exp−>exists(x | P(x)) holds if there exists an object x in exp for which P holds and exp−>one(x | P(x)) holds if there is *exactly* one such object. Since OCL expressions can be undefined, there exists an operation exp.oclIsUndefined(), which evaluates to true if the expression exp is not defined. The type of an object exp can be cast to a type C' using exp.oclAsType(C'). The operation allInstances() returns all instances of a class and its subclasses in the current state of the system, thus enabling statements about the universe of a certain class. For instance, Manager::allInstances() results in a set of all objects of kind Manager in the state in which the expression is evaluated.

2.3.1 Pragmatics of OCL.

Constraints stem from different sources: there may be legal restrictions that a system needs to obey; there may be company policies that grant privileges to certain kinds of customers; there may be technical restrictions on a system [Chen et al., 2006]; there may be security restrictions [Lodderstedt et al., 2002]; and there may be facts that are implied by common sense that cannot be expressed diagrammatically.

Whereas class models are intuitively accessible by non-IT-experts, OCL requires thorough knowledge of formal specification languages and object-oriented modeling. Therefore, although being an industry standard, OCL is not widely used in industry nowadays [Chiorean et al., 2005]. And even when it is used, it is difficult to write concise and correct specifications [Chiorean et al., 2004]. The usability of OCL is further restricted because frequently used derived operations such as the transitive closure for association ends are missing. Although they can be manually defined per model [Baar, 2003], this is time-consuming, error-prone, and requires information technology (IT) skills.

The concrete syntax of OCL is considered to contribute to the low acceptance of OCL in industry. Thus, several publications try to improve the syntax. For instance, a visual concrete syntax for OCL is proposed in [Bottoni et al., 2000], and a mathematical syntax is presented in [Süß, 2006] and [Brucker and Wolff, 2006].

There are several open-source tools that support model developers in creating constraint specifications, the most popular ones being (in alphabetical order) the Dresden OCL Toolkit [Dresden Technical University, 2007], the KeY project [Ahrendt et al., 2005], the OSLO project [Fraunhofer Fokus, 2007], the OCL Environment [Chiorean et al., 2003], and the USE tool [Gogolla et al., 2005]. Furthermore, there is an OCL plug-in for the Eclipse Modeling Framework (EMF) [Eclipse Foundation, 2007c] and an overview article on OCL tools [Álvarez et al., 2003].

Although these tools help developers in writing, type checking, and evaluating model states against constraints, the problem remains that OCL constraint specifications can contain *contradictory* constraints, especially when the specifications get large and complex. Since OCL is a first-order logic (FOL) with object-oriented extensions [Beckert et al., 2002], it is an *undecidable* language, i. e., it cannot be decided whether a given OCL specification is satisfiable or not. This leaves two options for finding contradictions, which we sketch briefly here and elaborate on in Section 2.6. First, semi-decision approaches or heuristic approaches can be used to automatically find contradictions. We are aware of two such approaches for OCL specifications, USE [Gogolla et al., 2005] and the Alloy Analyzer [Jackson et al., 2000] after transforming [Bordbar and Anastasakis, 2005] OCL to the Alloy specification language [Jackson, 2002]. Second, interactive theorem proving can be used to reason about constraint specifications. We are aware of OCL encodings for two theorem provers, HOL-OCL [Brucker and Wolff, 2006] for Isabelle/HOL [Nipkow et al., 2002] and one encoding [Kyas et al., 2005] for PVS [Owre et al., 1996].

2.4 Patterns

Besides augmenting models with textual constraints, models can be refined by applying patterns. In a nutshell, patterns are template solutions for recurring problems. With the success of the object-oriented development paradigm, patterns have gained increasing momentum in software engineering.

Patterns are developed by abstracting from solutions for concrete problems. A set $P = \{p_1, \ldots, p_n\}$ of patterns can be harvested (or "mined") according to the frequency of occurrence of a certain problem or to its importance. Typically, *anti-patterns* describe poor solutions that should be avoided.

Definition 8 (Pattern). *A pattern describes a generic solution to a recurring problem in a certain domain that can be reapplied to instances of the same problem.*

The most prominent publication, the Gang of Four (GOF) book on design patterns [Gamma et al., 1995], introduces a taxonomy of patterns for the construction of object-oriented software. Each pattern is presented with a name, classification, intent, structure, example, and other properties that describe its syntax, semantics, and pragmatics. Patterns have also become popular in other areas of software engineering, such as software architecture [Buschmann et al., 1996], formal specification [Dwyer et al., 1998, Ryndina, 2005], object-oriented analysis [Fowler, 1997], or workflow design [van der Aalst et al., 2003].

Syntactically, patterns are parameterizable model elements in this thesis. In Figure 2.8, we show an instance of the *bridge* pattern from [Gamma et al., 1995], which has four parameters, applied to our company model. Informally, the bridge pattern decouples an abstraction from its implementation such that the two can vary independently. To this end, it assigns roles to the elements used as its parameter. Figure 2.8

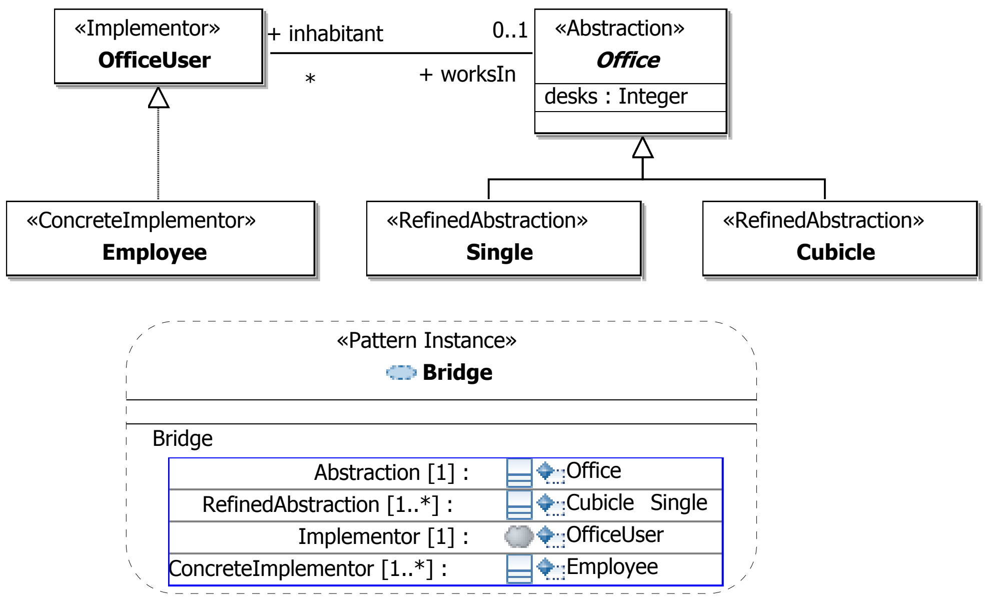


Figure 2.8: Bridge pattern applied to the company model.

shows a screenshot from the tool IBM Rational Software Architect (RSA), which we will use to implement the approach developed in this thesis. In RSA, pattern instances are represented as UML Collaborations, which are specialized using UML TemplateBindings (cf. [Object Management Group (OMG), 2006c]).

Patterns typically refine the models that they are applied to. The pattern semantics is often specified both informally and formally. The formal semantics of patterns can be either specified using formal pattern languages [Amálio et al., 2006, Elaasar et al., 2006], which allows for reasoning about patterns, also called meta-proofs [Amálio et al., 2006]. The results of the meta-proofs for each pattern hold for each instance of that pattern. Formal pattern semantics is also often specified in terms of a model transformation, i. e., when the models to which patterns have been applied are transformed, they get automatically refined.

2.4.1 Constraint Patterns.

A special class of patterns are *constraint patterns*, which we define as follows.

Definition 9 (Constraint Pattern). *A* constraint pattern *is a parameterizable constraint expression that can be instantiated to solve a class of specification problems.*

The concept of constraint patterns has been introduced for object-oriented programming [Horn, 1992] and recently been adapted in the literature for UML/OCL [Miliauskaitė and Nemuraitė, 2005, Ackermann and Turowski, 2006, Costal et al., 2006, Wahler et al., 2007]. Constraint patterns are also known as *idioms* [Ahrendt et al., 2005], and they are also important in other domains, e. g., constraint programming [Walsh, 2003].

Capturing and generalizing frequently used logical expressions is also performed in other domains than the refinement of class models. For instance, constraints play an important role in other domains, e. g., model transformations [Hauser et al., 2005], ontology modeling [Cranefield and Purvis, 1999] or model refactorings [Gogolla and Richters, 2002].

We capture the semantics of constraint patterns as an OCL template, i. e., a parameterizable OCL expression. To be more specific, we use these templates as *macros* because patterns are untyped. This allows model developers to use our template syntax not only for defining constraint patterns, but also for defining macros for frequently used OCL constructs.

The syntax of an OCL template starts with the keyword pattern followed by the name of the pattern and a set of typed parameters in brackets. This is followed by an equals sign and an arbitrary OCL expression in which the name of the formal parameters can be used. In the following, we define the *Singleton* pattern from [Gamma et al., 1995] using our template language.

```
pattern Singleton(element : Class) =
  element::allInstances()−>size() = 1
```

OCL templates can be used as first-class language elements of OCL. When they are instantiated, the formal parameters are replaced by the values of the actual parameters. We define instances of constraint patterns as follows.

Definition 10 (Constraint-Pattern Instance). *An instance ψ of a constraint pattern x is the value of $x(P)$ for a given set P of actual parameters.*

As an example, we instantiate the *Singleton* pattern to constrain the number of Cubicle offices in a model state to one. Note that in the following OCL statement, constraints *oneCubicle1* and *oneCubicle2* are semantically equivalent.

```
context Company
inv oneCubicle1: Singleton(Cubicle)
inv oneCubicle2: Cubicle::allInstances()−>size() = 1
```

As can be seen, constraint patterns are a concise means of hiding the syntactic and semantic complexity of OCL expressions and offering a unique name and uniform interface to the model developer. Although they help model developers avoid certain types of errors, e. g., confusing similar operators such as oclIsKindOf and oclIsTypeOf from the OCL specification, constraint specifications developed using constraint patterns can still be *inconsistent*. In the following section, we therefore elaborate on the notion of consistency in classical logic and in MDE.

2.5 Consistency

In this section, we discuss the notion of *consistency*. Since consistency is a well-known term in classical logics, we first give an overview of consistency for first-order logic (FOL) in 2.5.1. The term consistency is also increasingly used in MDE, but often with a different semantics. Therefore, give an overview of the different notions of consistency used in MDE in Section 2.5.2.

2.5.1 Consistency in Classical Logic.

We set the stage for consistency in the (rather complex) UML/OCL domain by looking at consistency in classical logic first. OCL is a three-valued first-order logic (FOL) with object-oriented extensions. Since statements in three-valued logics can be transformed into statements in two-valued logics [Behm et al., 1998, Abrial and Mussat, 2002] and OCL can be translated into FOL [Beckert et al., 2002], we begin by reviewing the consistency problem in FOL. We assume that the reader is familiar with the syntax and terminology of FOL as represented for instance in [Gallier, 1986].

The semantics $\phi_{\mathcal{M}}$ of a first-order formula ϕ is defined as a function $\phi_{\mathcal{M}} : s \rightarrow \{T, F\}$, where $\mathcal{M} = (\mathcal{U}, \mathcal{I})$ is a structure in which $\mathcal{U}$ is a set, called the *universe* and $\mathcal{I}$ is a mapping from syntax to semantics, called the *interpretation*. We require that the universe is nonempty, as otherwise universally quantified formulas trivially hold. Further, $s_\phi : V \Rightarrow \mathcal{U}$ is an *assignment function*, V is the set of variables in ϕ.

A formula ϕ is called *satisfiable* if there exists a structure $\mathcal{M}$ and an assignment s (in the following, we drop the index for s when the context ϕ is clear) such that $\phi_{\mathcal{M}}[s] = T$, also denoted by $\mathcal{M} \models^s \phi$. A structure $\mathcal{M}$ is called a *model* of ϕ, denoted by $\mathcal{M} \models \phi$ iff $\phi_{\mathcal{M}}[s] = T$ for every assignment s. OCL formulas are typically *sentences*, i. e., formulas without free (not bound by quantifiers) variables, postulating that the keyword self is universally quantified. The satisfiability of sentences is obviously independent of an assignment s. The notion of satisfiability is related to the notion of *consistency*, which is defined as follows.

Definition 11 (Consistency). *A set $\Phi = \{\phi_1, \ldots, \phi_n\}$ of sentences is* consistent *iff*

$$\exists \mathcal{M} . \mathcal{M} \models \phi_1 \wedge \ldots \wedge \phi_n \ ,$$

i. e., $\phi_1 \wedge \ldots \wedge \phi_n$ is satisfiable. Otherwise, Φ is inconsistent.

With an inconsistent set Γ of formulas, $\Gamma \rightarrow F$ is derivable [van Dalen, 1997], and from F, any statement can be derived, no matter if it holds or not.

A many-sorted FOL (MSFOL) is a FOL in which the terms have distinct types. Thus, the universe $\mathcal{U}$ in an MSFOL structure is defined as a function from a type to a set of objects, compared to FOL where the universe is merely a flat set of objects. The semantics of an MSFOL is defined by a many-sorted structure in which $\mathcal{U}(t)$ for each type t must be nonempty.

A simple example of an MSFOL language is $\mathcal{L}$, a language with one binary predicate R and one type α, which we define as follows.

$$\begin{aligned} \Gamma = \ \{ \ & (\forall x : \alpha).R(x, x), \\ & (\forall x : \alpha).(\exists y : \alpha).R(x, y), \\ & (\forall x : \alpha, y : \alpha).R(x, y) \wedge R(y, x) \rightarrow x = y\} \end{aligned}$$

This set is consistent if there exists a structure $\mathcal{M}$ that is a model for Γ. Such a structure is $\mathcal{M}_{\mathbb{N}}$. In this structure, the universe for the type α is $\mathbb{N}$ and the interpretation $R^{\mathcal{I}}$ of the predicate R is $\leq$, which denotes less-than-or-equal over the natural numbers $\mathbb{N}$.

$$\begin{aligned} \mathcal{M}_{\mathbb{N}} = \quad & (\mathcal{U}, \mathcal{I}) \text{ where} \\ & \mathcal{U}(\alpha) = \mathbb{N} \text{ and } R^{\mathcal{I}}(x, y) \equiv x^{\mathcal{I}} \leq y^{\mathcal{I}} \end{aligned}$$

Since $\mathcal{M}_{\mathbb{N}} \models \Gamma$, Γ is consistent.

2.5.2 Consistency in MDE.

The term *consistency* is used with different meanings in MDE. In general, one needs to distinguish between *inter-model* consistency and *intra-model* consistency. In the following, we define these notions.

Inter-Model Consistency. In MDE, inter-model consistency with respect to a predicate P denotes that two or more model states satisfy P. P is defined on their respective metamodels and expresses relations between elements in the respective models.

Definition 12 (Inter-Model Consistency). *Two model states $m_1 : M_1, m_2 : M_2$ are* inter-model consistent *wrt a relation $R : M_1 \times M_2$ iff $(m_1, m_2) \in R$.*

For example, a UML class diagram c and a UML sequence diagram s are considered inter-model consistent if all classes and operations used in s are defined in c. Inter-model consistency is the subject of numerous publications, e. g., [Emmerich et al., 2003, Jonckers et al., 2003, Küster, 2004, Sabetzadeh et al., 2007].

Intra-Model Consistency. In contrast, intra-model consistency of a model M denotes that M can be instantiated, i. e., there exists at least one state for a given model. It thus corresponds to the notion of consistency in classical logic.

Definition 13 (Intra-Model Consistency). *A model M is* intra-model *consistent iff there exists a state τ of M.*

For example, a class model is intra-model consistent if there exists at least one model state for it.

Since we introduce an approach to developing consistent constraint specifications in this thesis, we focus on intra-model consistency of one important model type of UML: class models annotated with invariants in the OCL. Since OCL can be considered an MS-FOL in which the classes in the corresponding UML model constitute the types, we could apply the "classical" notion of consistency to UML/OCL models. However, several peculiarities of UML/OCL must be taken into account such as its hierarchical type system, abstract classes, or statements about the universe of a certain class using the OCL operator allInstances(). Thus, the classical notion of consistency is not directly applicable to UML/OCL models. Therefore, distinct notions of consistency are required for intra-model consistency of UML/OCL models, which we discuss in Chapter 5.

Consistency is also important in other development approaches. For the Vienna Development Method (VDM), a formal approach to system development [Jones, 1990], various consistency properties are discussed in [Damm et al., 1991]. In [Aichernig and Larsenz, 1997], it is explained how proof obligations can be generated that ensure the consistency of VDM specifications.

The Java Modeling Language (JML) [Leavens et al., 2006] is an interface specification language designed to specify Java classes and interfaces. In [Darvas and Müller, 2006], JML specifications are transformed into axioms. In order to prove the consistency of these axioms, the existence of a witness has to be proven in order to employ the corresponding axiom.

2.6 Interactive Theorem Proving

Since OCL is based on FOL [Beckert et al., 2002], it is an undecidable language. Thus, properties of OCL statements such as consistency cannot be proven automatically. However, interactive theorem provers assist in carrying out proofs by providing proof tactics, which automate various steps in proofs. In Section 2.6.1, we give an overview of Isabelle/HOL and introduce the concepts of interactive theorem proving used in this thesis.

For carrying our proofs for OCL specifications, HOL-OCL [Brucker and Wolff, 2006] offers an embedding of OCL into higher-order logic (HOL) and an OCL proof environment for the interactive theorem prover Isabelle/HOL [Nipkow et al., 2002]. This allows us to formally carry out consistency observations of UML/OCL models and to express the semantics of constraint patterns using function types. In Section 2.6.2, we give an overview of HOL-OCL.

2.6.1 Isabelle/HOL.

Isabelle/HOL [Nipkow et al., 2002] is an interactive theorem prover that combines functional programming and higher-order logic (HOL). Thus, types and functions can be defined and theorems about them can be proven. Interactive theorem provers offer partial automation of proofs by using advanced rewriting techniques and proof tactics. In the following, we explain the concepts of Isabelle/HOL that we are using in the remainder of this thesis by means of an example system and proof.

We model a system that allows us to add natural numbers. We first define a recursive datatype STerm that represents terms of our system. STerm has two constructors, Atom and Plus. Atom has one parameter of type nat, which is predefined in Isabelle/HOL and represents natural numbers. The second constructor, Plus, is composed of two terms of type STerm.

```
datatype STerm = Atom nat |
                 Plus Sterm Sterm
```

Next, we define a function evaluate, which maps a term of type STerm to a natural number. The function is defined using primitive recursion. In Isabelle/HOL, one equation must be defined for each constructor of the recursive datatype used as argument for the primitive recursive function. Thus, we specify one equation for Atom and one for Plus.

```
consts evaluate :: ”STerm ⇒nat”
primrec
  ”evaluate (Atom n) = n”
  ”evaluate (Plus x y) = (evaluate x) + (evaluate y)”
```

Having defined a function on our recursive datatype STerm, we can now prove a simple statement about the function. The following lemma states that if we add a natural number to a positive number, the result will also be positive.

```
lemma ”(0 < x) ⟹0 < (evaluate (Plus (Atom x) (Atom y)))”
```

After specifying a lemma, Isabelle/HOL displays a proof goal that corresponds to the lemma. To prove this goal, we have to invoke one of the proof tactics that Isabelle/HOL provides. A proof tactic in Isabelle/HOL is a sequence of rewriting rules that can be applied to proof goals, e. g., induction, case splits, or simplification. For our example theorem, it is sufficient to invoke simplification, which replaces the elements in the conclusion of our theorem by their definitions until the resulting proof goals are axioms. Proof tactics are invoked with the keyword **apply**; the simplification tactic is called simp.

```
apply simp
```

After invoking a proof tactic, a set of new proof goals is displayed that each need to be proven. In our example, this set is empty, which means that all goals have been proven and thus, the original lemma. Proofs are completed by the following statement.

```
done
```

Alternatively, if one or more proof goals cannot be established, a proof can be abandoned by the keyword **oops**.

Isabelle/HOL also supports *type variables* and *axiomatic type classes*. The former can be used to define generic functions. For example, the signature of a generic function that computes the inverse of an arbitrary type can be defined as follows.

```
consts inverse :: "'a ⇒ 'a"
```

In this definition, 'a denotes an arbitrary type. Types can belong to *type classes*. For example, it makes sense to refine the inverse function such that it is not applicable to arbitrary types, but to types only that belong to some class of invertible types. We define a type class invertible and refine inverse as follows.

```
axclass invertible < type

consts inverse :: "('a :: invertible ) ⇒ 'a"
```

Note that in above **consts** declaration, we need to specify only once that 'a belongs to the type class invertible .

Another concept that we use in this thesis are lambda expressions for defining functions. For example, we can model the successor function on natural numbers using a lambda expression as follows.

```
constdefs
 suc :: "nat ⇒ nat"
 "suc == λ x. (x+1)"
```

Among other proof assistants for interactive theorem proving such as PVS [Owre et al., 1996] or Coq [Huet et al., 2008], we choose Isabelle/HOL because it is used as the basis for HOL-OCL, a formal semantics and proof environment for UML/OCL that we use in various parts of this thesis.

2.6.2 HOL-OCL.

HOL-OCL [Brucker and Wolff, 2006] defines a formal semantics for UML/OCL and provides support for formal reasoning about object-oriented models. This is achieved by representing UML/OCL as a shallow embedding into the interactive theorem prover Isabelle/HOL, i. e., the concepts of UML/OCL are defined in terms of the concepts of the meta-language HOL.

In this section, we briefly summarize the formal semantics of UML/OCL as presented in [Brucker and Wolff, 2006]. In particular, we present the main properties of the underlying object store and the notion of validity of an OCL formula.

2.6.2.1 Object Store.

In the context of this thesis, we use OCL for constraining class models. Therefore, we need a formal semantics of class models, i. e., a model of an object store, in addition to a formal semantics of OCL [Brucker, 2007, Object Management Group (OMG), 2006b]. In HOL-OCL, the *object store* provides the core notion of object-oriented data structures, e. g., a formalization of classes and instances, including concepts such as inheritance and subtyping. In particular, the object store provides the formal semantics of path expressions, i. e., expressions for navigating through a concrete object structure. Moreover, HOL-OCL allows for reasoning about infinite sets of objects, which will become important when we investigate consistency concerns of OCL specifications in Chapter 5. The OCL operator size() is defined to map infinite sets to $\perp$. In this section, we present a simplified version of the HOL-OCL formalization presented in [Brucker, 2007].

The core idea underlying the formalization of an object store is to use HOL and represent classes as tuples $(X \times Y)$. In this HOL representation, an abstract type oid is assumed, which represents the object identifier, i. e., a reference to an object. For each class C, an abstract type C_{tag} is introduced, which is an abstract datatype that guarantees the uniqueness of this construction: Even if two classes have the same attribute types, they can be distinguished by their tag types. The class attributes are either of a basic type, e. g., Integer or String, a reference to another object (oid), or a collection thereof, e. g., oid Set. In the object model of HOL-OCL, a common supertype of all objects can be represented without loss of generality because a common supertype can always be added to a given class structure without changing the overall semantics of the original object model. Such a supertype is, for example, defined in OCL as OclAny or in Java as Object.

For example, we can represent the type Manager in our HOL encoding as a tuple in which the type hierarchy from OclAny to Manager is represented. For each type in the hierarchy, its structural features are represented, e. g., Employee has two attributes and two associations to other classes, as can be seen in Figure 2.5. Class Manager has three attributes and one reference to another class. In our HOL encoding, the type of such a reference is oid Set. The common supertype OclAny has one structural feature only, which is the object identifier oid. The tuple reads as follows in HOL.

```
Manager := ((OclAny_tag ×oid) ×
            (Employee_tag ×oid Set ×oid Set ×String ×Integer) ×
             (Manager_tag ×oid Set ×Integer ×Integer ×Boolean))
```

In HOL-OCL, types are parameterized by the system state τ and belong to the type class bot. The keyword VAL denotes single values, whereas Set indicates collections. On top of this representation, the usual operations like type casts, constructors, or attribute accessors can be defined in a type-safe manner. For details, we refer the reader to [Brucker, 2007]. The current set of objects that is represented by an object state characterizes the system state. As defined in Section 2.2.2, we call a specific configuration of the object store, i. e., a certain instance of the model, the state τ of the system.

2.6.2.2 Constraining an Object Store: Invariants and Types.

By combining an object store with an object-oriented constraint language, e. g., OCL, one can restrict the set of valid object structures semantically, i. e., specific states are ruled out if they violate the constraint specification. One important part of an OCL specification are class invariants, which constrain the set of valid instances of a class. In HOL-OCL, an invariant inv_X of a class X is a function that maps an object x of class X to a truth value or to an undefined value $\perp$, depending on the system state τ in which inv_X is evaluated.

The notion of class type is extended by the requirement that an instance of a class has to fulfill the invariants of this class. Therefore, HOL-OCL provides semantic *type sets*, based on a co-recursive construction described in [Brucker, 2007]: For each class X, HOL-OCL defines the set $\mathfrak{C}_X$ of *valid* objects, i. e., all objects that fulfill the class invariant.

The following property holds for our construction: If an element x is in the set of valid objects of its class X in a state τ, then x satisfies the invariants defined on X.

$$\frac{\tau \vDash x \in \mathfrak{C}_X}{\tau \vDash \text{inv}_\text{X}\, x}, \tag{2.1}$$

where τ denotes the current system state and the term $\tau \vDash \phi$ denotes that the system state τ satisfies the formula ϕ. Informally, this rule means that all objects in the semantic type set of a class have to satisfy the invariant of this class. From this rule, the following properties follow directly: If an object x is of a given type X, then it satisfies its invariants (Equation 2.2). If no object satisfies the invariants of some class X, then type set of X is empty (Equation 2.3).

$$\frac{\tau \vDash x.\mathsf{oclIsTypeOf}(X)}{\tau \vDash \mathrm{inv}_{\mathrm{X}}\ x} \quad , \quad \frac{\tau \vDash X :: \mathsf{allInstances}()\mathsf{-}\mathsf{>}\mathsf{includes}(x)}{\tau \vDash \mathrm{inv}_{\mathrm{X}}\ x} \quad , \tag{2.2}$$

$$\frac{\neg\exists x. \tau \models \mathrm{inv}_{\mathrm{X}}\, x}{\tau \models \mathfrak{C}_X = \emptyset} \quad \text{, and} \quad \frac{\tau \models \mathfrak{C}_X = \emptyset}{\tau \models \forall x. x.\mathsf{oclAsType}(X).\mathsf{oclIsUndefined}()} \quad . \tag{2.3}$$

We will refer to these properties in Chapter 5 when we define consistency notions for UML/OCL models.

Chapter 3

Constraint Elicitation and Specification

As shown in Section 2.3, class models must typically be refined with textual constraints to increase their maturity levels. To this end, the model developer must thoroughly analyze and elicit constraints. Such an elicitation is typically performed using domain analysis [Prieto-Díaz, 1990, Hjørland and Albrechtsen, 1995], in which the modeled system is manually analyzed and constraints are added by the model developer based on domain knowledge. However, the model developer may not detect all causes for low maturity, which can cause serious problems in the MDE process because the generated code may cause runtime exceptions or worse, be wrong.

In this chapter, we present an approach to constraint elicitation that is complementary to domain analysis. Whereas constraints elicited by domain analysis are specific to the respective domain, our approach focuses on eliciting domain-independent constraints, i. e., constraints that occur in any kind of class model. To this end, we identify limitations of the expressiveness of class models that require refinement and present a remedy against these anti-patterns in the form of textual constraints in OCL. Subsequently, we capture these limitations in the form of anti-patterns. This enables an automatic constraint elicitation that complements domain analysis.

This chapter is structured as follows. By analyzing the MOF meta-model, we identify typical limitations of the expressiveness that require refinement, present examples to illustrate the potential dangers of these limitations, and show how they can be remedied by refining class models with textual constraints in Section 3.1. In Section 3.2, we show how to capture such limitations as *anti-patterns* that can be used for automatic constraint elicitation.

3.1 Limitations of the Expressiveness of Class Models

In UML or MOF, model states can only be partially constrained by means of the diagrammatic class-modeling languages. The meta-model for class models (cf. Section 2.2) allows model developers to define classes, attributes, and operations. Properties and operations have types, and they are multiplicity elements, i. e., they can be sets with a predefined minimum or maximum size. However, the meta-model is not expressive enough to express details that frequently occur in systems and need to be expressed in the model. Such limitations of the expressiveness typically require refinement with textual constraints.

In this section, we show by example how to find limitations of the expressiveness of class models. For each limitation, we show a set of problems that it can cause. Subsequently, we specify a textual constraint in OCL that prevents these problems. We have drawn the knowledge about these limitations from our experience in class modeling. In addition, the case studies in Chapter 8 provide evidence that these limitations frequently pose problems and can only be remedied by enriching the respective class model with textual constraints.

3.1.1 Unrestricted Multiplicity Elements.

As shown, the multiplicities of properties can only be roughly constrained in a diagrammatical way in class models. In the following, we present two cases in which this limited way of defining multiplicities causes problems.

Multiplicities depending on an attribute value. In class Manager, we modeled an attribute headCount, which denotes the maximum number of employees that a manager can employ. Meta-models such as MOF or UML do not provide means to specify that the number of employees in the employment relation depends on the value of headCount. Therefore, the instance in Figure 3.1 is valid: Although anna has a maximum head count of one, she can employ two employees.

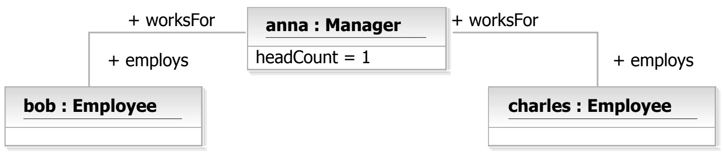


Figure 3.1: Two employees despite a maximum head count of one.

Since the instance shown in Figure 3.1 cannot be excluded in terms of the meta-model of class models, an OCL constraint is required that restricts the employment relation depending on the value of the headCount attribute. If the company model is annotated with the following invariant *headCountRestriction*, the instance shown above is invalid.

```
context Manager
inv headCountRestriction: self.employs−>size() <= self.headCount
```

Context-unaware association semantics. Associations represent relations between classes. Often, associations are created with a certain semantics in mind, but the semantics is not specified. The relation between Employee and Manager in Figure 3.2 shows variables as lower and upper multiplicity bounds. The values for these variables determine the semantics of the relation. The relation can be a function ($x_1 = 0, y_1 = *, x_2 = 0, y_2 = 1$), a *total* function (0..* / 1..1), an *injective partial* function (0..1 / 0..1), an *injective total* function (0..1 / 1..1), a *surjective partial* function (1..* / 0..1), a *surjective total* function (1..* / 1..1), or a *bijective* function (1..1 / 1..1).

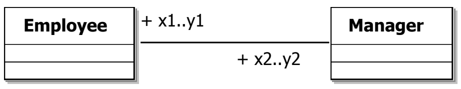


Figure 3.2: A generic binary association between Employee and Manager.

The semantics of associations can be specified by assigning values for the multiplicities of each property involved in the association. However, if the semantics of an association depends on other elements in the model instance, e. g., the value of an attribute, this cannot be expressed in terms of the diagrammatic class model. The following example illustrates this problem: In the model in Figure 2.5, the association employs from Manager to Employee is surjective, i. e., every employee needs to work for at least one manager. This is a problem, since the CEO of the company should have no manager. Thus, surjectivity for the employment relation is only required for employees who are not the CEO. However, this cannot be expressed in MOF and thus requires a textual constraint, which we formalize in constraint *hasManager* as follows.

```
context Manager inv hasManager:
 if not self.isCEO
then self.employs.allInstances()−>forAll ( y |
     self.allInstances()−>exists( x | x.employs−>includes(y)))
else true
endif
```

Note that the multiplicity of worksFor in the company model should be relaxed from 1..∗ to ∗ to avoid contradictions with *hasManager*.

3.1.2 Insufficiently Typed Associations.

Figure 3.3 shows an instance of our example model where anna, a Manager, works in a cubicle, while charles, an Employee, works in a single office. This instance is valid because it conforms to the meta-model.

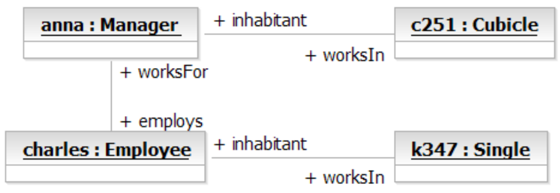


Figure 3.3: Manager and employee inhabiting “inappropriate” offices.

However, a company policy may have the requirement that only managers may work in single offices. This constraint is violated by the instance in Figure 3.3. Therefore, a textual OCL constraint is necessary that restricts the usage of subclasses, which we specify as follows.

```
context Single
inv onlyManagers: self.inhabitant−>forAll(x | x.oclIsTypeOf(Manager))
```

3.1.3 Unrestricted Reflexive Associations.

Reflexive associations are binary associations whose both ends belong to the same class. In Figure 3.4, we illustrate three ways in which reflexive associations can occur. In its simplest form, a reflexive association connects one class to itself such as inverse for the class Color. A less obvious reflexive association is worksFor for the class Manager: Since Manager is a special kind of Employee, it inherits the worksFor association. Thus, managers can work for other managers or for themselves. The third example shows a reflexive path: Starting from the class Operation, the path output.input leads back to this class. We subsume reflexive paths in our notion of reflexive associations.

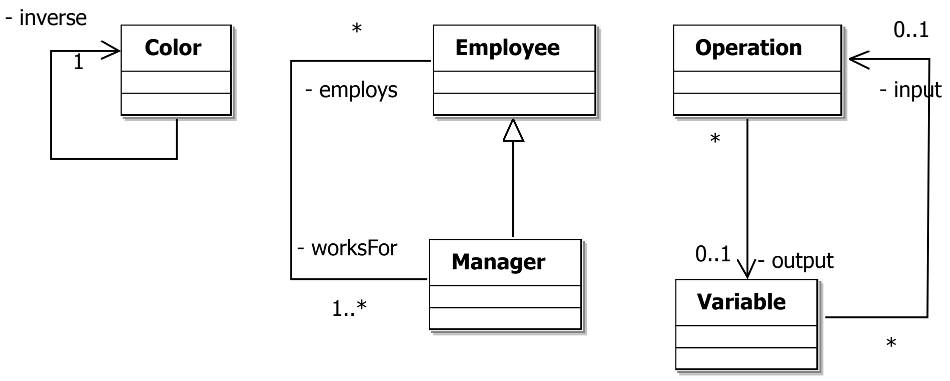


Figure 3.4: Examples of reflexive associations.

Reflexive associations are an important means for modeling systems, since the concept of reflexivity is ubiqituous in many systems: The mother of a human being is a human being, the inverse of a color is a color, and the superior of a manager is a manager.

In general, reflexive associations need to be treated with care because they allow objects to be related to themselves, which corresponds to the notion of reflexive relation in mathematics. Often, additional constraints are necessary to rule out invalid relations, as in the following example. Although the successor of a natural number is a natural number, the Peano axioms, which can be considered a meta-model for natural numbers, ensure that the set of natural numbers is infinite and the successor relation is acyclic.

Unconstrained reflexive associations can cause a low maturity level for three reasons. First, they enable cycles in the object graph, second, they allow an arbitrary number of objects to be related in a chain, and third, they allow for so-called diamond configurations. In the following, we point out these problems in detail and show how textual constraints are used to remedy the expressive deficiencies of graphical modeling languages.

Cycles. Reflexive associations can cause cycles in the object graph. Cycles may be desired in certain systems: For instance, in usual color spaces, the inverse of the inverse of a color is the color itself. However, such cycles are in many systems: a person cannot be the mother of her mother, and a natural number is not the successor of itself. The reflexive association worksFor can cause cycles in instances of the company model. We illustrate such a cycle in Figure 3.5.

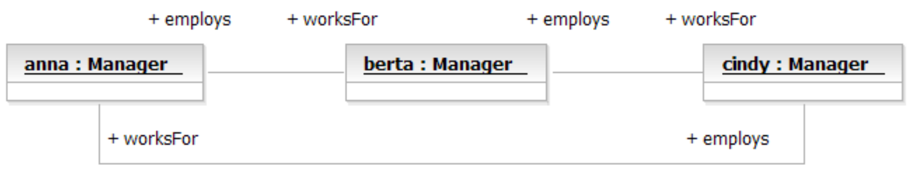


Figure 3.5: A cyclic management relation.

The model developer needs to be aware that reflexive associations can cause cycles in object graphs and he needs to carefully assess whether cycles are valid structures in the system that is modeled. If not, cycles must be excluded using OCL constraints.

For the definition of such constraints, an operation to compute the transitive closure of an operation is required. Since there is no such operation in OCL (cf. Section 2.3), the transitive closure of each association needs to be manually defined. In the following, we define a transitive closure operation for the worksFor association and state an invariant that the context object may not be a member of the transitive closure of its worksFor association.

```
context Manager

def: closureWorksFor(S:Set(Manager)) : Set(Manager) =
     worksFor->union((worksFor - S)->
       collect (m : Manager | m.closureWorksFor(S->including(self)))->asSet())

inv noCycles: self.closureWorksFor(Set{})->excludes(self)
```

In the definition of the transitive closure operation, we use a parameter S to ensure termination of the operation. S represents the set of objects for which the transitively reachable objects have already been calculated. On each recursive invocation of the operation, S grows by one element. Thus, the set worksFor − S will eventually be empty and the operation terminates, provided that there is a finite number of objects in the respective model state. We added this parameter after we found out that the evaluation of a simple declarative definition of the transitive closure, i. e., worksFor−>union(worksFor.closureWorksFor()), would not terminate in most current OCL tools.

Arbitrary path lengths. Another problem with reflexive associations is that navigation paths in model states can be arbitrarily long. For certain application domains, the maximum path length needs to be restricted. For instance, Figure 3.6 shows an instance of the company model with eight hierarchy layers.

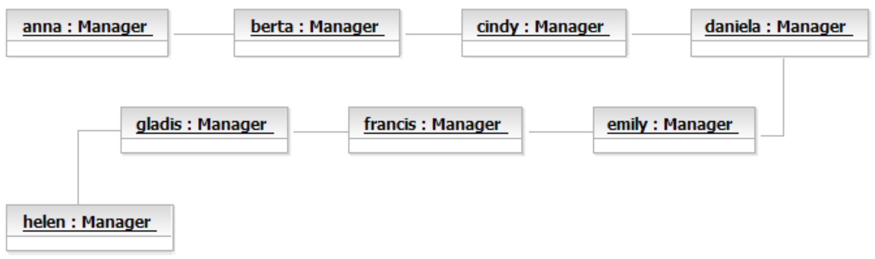


Figure 3.6: A company with eight management levels.

Such configurations are not valid for most systems modeled and should not be allowed. However, the length of such paths cannot be restricted in terms of the meta-model for class models and thus needs textual constraints that require recursive queries as introduced in Section 2.3. The following constraint restricts the path length of the worksFor association to 5. It consists of two parts: a definition for the recursive query and the actual invariant, which uses the previously defined query.

```
context Manager

def: pathDepthWorksFor(max: Integer, counter: Integer): Boolean =
  if  (counter > max or counter < 0 or max < 0) then false
  else  if  (self.worksFor->isEmpty()) then true
        else  self.worksFor->forAll(m:Manager|m.pathDepthWorksFor(max, counter+1))
        endif
  endif

inv smallHierarchy: self.pathDepthWorksFor(4,0)
```

Diamond configurations. Reflexive associations can cause a third kind of undesired configuration, namely *diamonds*. Diamond configurations have been known for a long time [Newman, 1942] and have become known as “Nixon diamonds” in nonmonotonic reasoning [Reiter and Criscuolo, 1981] or “deadly diamonds of death” in object-oriented programming languages with multiple inheritance such as C++ [Martin, 1998].

Definition 14 (Diamond Configuration). *A model state τ of a model M contains a diamond configuration between objects o_1 and o_4 iff there exist objects $o_2, o_3 \in \tau$ and links $(o_1, o_2), (o_1, o_3) \in \tau$ and there are (direct or transitive) links from o_2 to o_4 and from o_3 to o_4.*

In our company model, the reflexive association worksFor can cause diamond configurations between managers as shown in Figure 3.7: daniela has two managers berta and cindy, who work for the same manager anna. Such a configuration can cause the following problem: If anna tells berta to fire all employees and tells cindy to keep all employees, it is not specified what happens to daniela, who works for both berta and cindy. Therefore, diamond configurations must be treated with special care and even may have to be excluded in many cases.

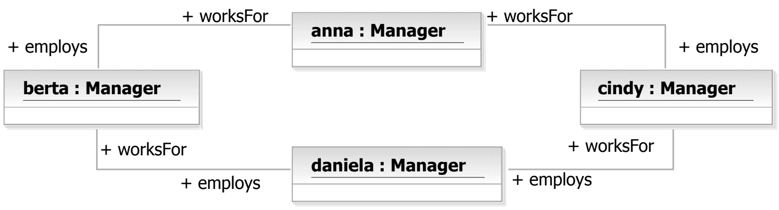


Figure 3.7: Diamond configuration of managers.

Such a configuration can be excluded with the following constraint *noDiamond* in which we re-use the previously defined operation closureWorksFor().

```
context Manager inv noDiamond:
  self.worksFor−>exists(m1,m2 | m1−>closureWorksFor(Set{})−>intersect(
                                  m2−>closureWorksFor(Set{}))−>notEmpty()
                      implies  m1=m2)
```

3.1.4 Missing Unique Identification.

Figure 3.8 shows a valid instance of the company model that is likely to cause integrity problems when stored in a data base because employees e1 and e2 are indistinguishable by their name.

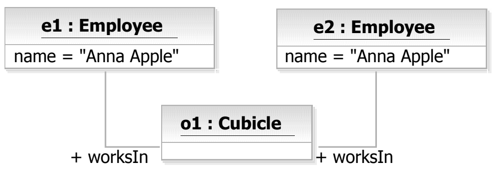


Figure 3.8: Two employees with the same name sharing an office.

Using the OCL operation isUnique, we can textually specify the tuple (name,worksIn) to be the primary key for class Employee. This renders above example state invalid. The constraint reads as follows.

```
context Employee
inv uniqueness: self.allInstances()−>isUnique(e|Tuple(x=e.name,y=e.worksIn))
```

3.1.5 Unrestricted Attribute Values.

The values of attributes of one or more classes cannot be related to each other in terms of the class modeling language. In this subsection, we illustrate two examples for why this lack of expressiveness causes low maturity of class models. We distinguish between *simple* and *complex* relations of properties.

Simple relations of attribute values. Two properties can be related by a binary operator such as less-than (<). However, such relations cannot be modeled in terms of the MOF meta-model. Figure 3.9 shows an instance of the company model that conforms to the meta-model although the employee charles has a higher salary than his manager anna, which may not conform to their company's policy.

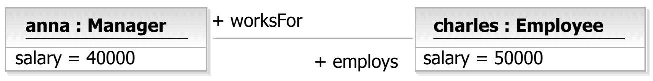


Figure 3.9: An employee has a higher salary than his manager.

To exclude such instances, the following OCL constraint *higherSalary* needs to be added to the company model. The constraint requires that the salary of a manager is higher than the salary of each employee.

```
context Manager
inv higherSalary: self.employs−>forAll( e | self.salary > e.salary )
```

Complex relations of attribute values. In our example world, the budget of a manager is used to pay the salary of the manager's employees. Therefore, the budget must be at least the sum of the salaries of all employees whom a manager employs. However, this fact cannot be expressed in terms of MOF, and therefore, the instance in Figure 3.10 is a valid instance of the company model, although anna cannot pay the full salaries for bob and charles.

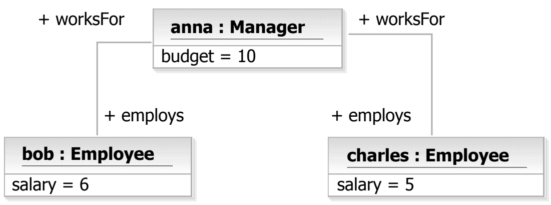


Figure 3.10: Sum of employees' salaries is higher than the budget.

In order to exclude the instance from Figure 3.10, we annotate the company model with the following invariant, *budgetRestriction*.

```
context Manager
inv budgetRestriction: self.employs.salary−>sum() <= self.budget
```

In the next section, we explain how the limitations presented in this section can be captured as anti-patterns.

3.2 Capturing Limitations as Anti-Patterns

In this section, we describe how the problems found in the previous section can be abstracted to *anti-patterns*, which enables an automatic elicitation of these limitations. For each anti-pattern, we describe the problem that its occurrence can cause in an abstract way. Furthermore, we assess the *severity* of each pattern, i. e., the gravity of the problem that the respective pattern causes.

3.2.1 Unrestricted Multiplicity Elements.

Since Property is a MultiplicityElement (Figure 2.3), properties have a lower and an upper bound for the multiplicity of the association end they denote, i. e., the cardinality of the domain of the relation. The lower bound reflects the minimum number and the upper bound reflects the maximum number of objects that need to be in the domain of the relation. As shown in Figure 2.3, the lower and upper boundary can be either a natural number or arbitrarily large, represented by the symbol $*$.

The upper multiplicity of an association is often unbounded ($*$) because in most systems, the number of elements in a relation is not restricted to a fixed literal value. For instance, we used an unbounded multiplicity for all associations in the company model (Figure 2.5), except for the property worksIn of Employee, because an employee can be related to at most one office in our system.

However, an unspecified number of elements in a relation can potentially cause a low maturity level of the model. In the company model, the employment relation is an example of low maturity: It allows managers to employ any natural number of employees and every employee may work for arbitrarily many managers (but at least one).

We summarize this anti-pattern in Table 3.1.

Unrestricted Multiplicity Elements	
Description	This pattern occurs when the upper multiplicity bound of a multiplicity element is unbounded, also denoted by an asterisk (*).
Problem	Multiplicity bounds often depend on the model state, e. g., the value of a certain attribute. However, this cannot be expressed diagrammatically in class models.
Severity	Medium – Occurrences of this pattern can cause a relation of one object with too many objects.

Table 3.1: The *Unrestricted Multiplicity Elements* anti-pattern.

3.2.2 Insufficiently Typed Associations.

According to the meta-model in Figure 2.3, the type of a property can be a class. This allows model developers to create associations from one class to any other class, even to a class that has specialized subclasses. Thus, any subclass of the superclass can take the role of the superclass in the association. However, in some scenarios, this is unwanted but cannot be prevented by means of the class model syntax.

We summarize this anti-pattern in Table 3.2.

Insufficiently Typed Associations	
Description	This pattern occurs when an association between two classes is defined and either of them has at least one subclass that specializes it.
Problem	In this case, the association needs to be refined such that associations with certain subclasses are required or prohibited.
Severity	Medium – Occurrences of this pattern can cause underspecified relations between objects.

Table 3.2: The *Insufficiently Typed Associations* anti-pattern.

3.2.3 Unrestricted Reflexive Associations.

In class models, reflexive associations can cause a low maturity level if unconstrained. First, they enable cycles in the object graph, second, they allow an arbitrary number of objects to be related in a finite chain, and third, they allow for so-called diamond configurations, i. e., there exists more than one path between two objects whose length is greater than one.

The consequences of cyclic dependencies and diamond configurations between objects for the generated code are severe. Cyclic dependencies can cause nonterminating computations and diamond configurations can cause synchronization problems. For example, in formal proof environments such as Isabelle/HOL [Nipkow et al., 2002], it must be explicitly proven that recursive definitions terminate. In Section 8.2, we introduce a model in which these problems are illustrated.

We summarize this anti-pattern in Table 3.3.

Unrestricted Reflexive Associations	
Description	This pattern occurs when a class C is related to itself, i. e., the graph of the class model in which classes denote nodes and associations denote edges contains a cycle. There are two causes for this. First, there can be an association a of which both ends connect to C. Second, there can be a sequence of associations and a corresponding path expression $x_1.x_2.\ldots.x_n$ that relate C indirectly to itself.
Problem	Reflexive associations can cause problem such as cyclic dependencies between objects if unrestricted.
Severity	High – Occurrences of this pattern can cause nonterminating computations in the generated code.

Table 3.3: The *Unrestricted Reflexive Associations* anti-pattern.

3.2.4 Missing Unique Identification.

Our example model of a company in Figure 2.5 is a data model. It is usually required for data models that objects can be uniquely identified, i. e., they must have a primary key. In MOF, a property of a class can be made a unique identifier by setting its isID attribute to true. However, only one property of a class may be a unique ID [Object Management Group (OMG), 2006a], which excludes primary keys that are composed of several properties. In UML class models, it is not possible to assign such a unique-key property to class attributes [Muller, 1999]. Instead, defining such a property requires using UML profiles [Gornik, 2002] or OCL constraints [Demuth and Hussmann, 1999].

In our example, the name of an Employee can be made a primary key in the company model in terms of MOF. However, if we want to compose the primary key from the properties name and worksIn, we need to add a textual constraint to the model because composed keys cannot be modeled in terms of MOF. We summarize this anti-pattern in Table 3.4.

Missing Unique Identification	
Description	This pattern occurs when a class C has a nonempty set of attributes, but none of them is assigned to be the primary key of C.
Problem	Although objects typically have unique references, it is possible for two or more objects of the same class to have identical values for all of their attributes. This can make them indistinguishable and lead to runtime problems in the system model.
Severity	Medium – Occurrences of this pattern can compromise the integrity of model states.

Table 3.4: The *Missing Unique Identification* anti-pattern.

3.2.5 Unrestricted Attribute Values.

In system models, properties of the same class or of different classes are related, i. e., the value of one property depends on the value of other properties. Although such relations are common, the meta-model for class models does not provide any means to express such relations. We summarize this anti-pattern in Table 3.5.

Unrestricted Attribute Values	
Description	This pattern occurs when attributes of the classes in a model are not related to each other although their values depend on each other in the modeled system.
Problem	Attributes are often not independent of each other. However, relations between attributes cannot be expressed in class modeling languages. Thus, such relations cannot be expressed diagrammatically by means of the meta-model.
Severity	Medium – Occurrences of this pattern can cause undesired model states.

Table 3.5: The *Unrestricted Attribute Values* anti-pattern.

3.3 Summary

Diagrammatic languages such as defined by MOF or UML have been successfully used in various development projects. However, model developers must be aware that diagrammatic languages alone are not sufficient for developing class models with high maturity levels. We have shown by example that the limited expressiveness of the meta-model for class models requires refinement of class models with textual constraints and how such limitations can be captured as anti-patterns.

The contribution made is twofold. First, it is important for model developers to be aware of such limitations and our observations may help to identify further limitations. Second, capturing the limitations as anti-patterns allows us to automatically match them against model states. As part of the tool support for our approach, we have developed a plug-in that searches class models for occurrences of the anti-patterns and displays the results to the model developer. In Chapter 7, we elaborate on the details of our tool support.

It is important to mention that in general, the occurrence of an anti-pattern does not imply a problem in the respective model. However, occurrences of anti-patterns denote *potential* problems and should be examined by the model developer if they actually pose problems.

We have shown how OCL constraints can be used to remedy the anti-patterns that we have identified to increase the maturity level of class models. However, writing correct constraint specifications for class models is a time-consuming task that requires significant amount of expertise [Chiorean et al., 2005]. In addition, it comprises numerous repetitive tasks because constraints against several occurrences of one anti-pattern share a similar structure. As explained in Section 2.4, patterns are generally useful for recurring problems. In particular, constraint patterns (cf. Section 2.4.1) are useful for solving recurring specification problems. Therefore, we present in the following chapter how constraint patterns can be used as a concise means to prevent anti-patterns from occurring in model states. For each of the anti-patterns introduced in this chapter, we will present one or more constraint patterns that can be used to remedy the respective anti-pattern.

Chapter 4

Specifying Constraints Using Patterns

In the previous chapter, we have identified recurring problems in class models as anti-patterns that require refinement with textual constraints. However, developing constraint specifications is not an easy task: Besides theoretical and practical arguments that point out various deficiencies of OCL [Chiorean et al., 2005, Süß, 2006, Cabot, 2006, Brucker et al., 2006], one important aspect needs to be taken into account: Class models can express complicated relationships, including subtyping, reflexive relations, or potentially infinitely large instances, and constraining such facts requires addressing this complexity.

Furthermore, constraint specifications need to be maintained together with the model and be adapted once the model changes. This usually results in additional time-consuming coding and debugging phases, especially in refactorings [Correa and Werner, 2004, Pretschner and Prenninger, 2005, Markovic and Baar, 2005] where models undergo frequent changes and the attached constraints need to be kept consistent with new versions of the model.

Constraint patterns promise to shorten the development time and decrease the error rate for constraint development by offering predefined templates. However, existing publications on constraint patterns [Miliauskaitė and Nemuraitė, 2005, Ackermann and Turowski, 2006, Costal et al., 2006] focus on very specific refinement problems and are thus not flexible enough to be used for expressing more general constraints and in particular, expressing the constraints from Section 3.1, which we used to remedy the previously identified anti-patterns. Therefore, we introduce the concept of *composable* constraint patterns that allows model developers to logically connect different constraint patterns and, thus, to specify complex constraints in terms of constraint patterns.

This chapter is organized as follows. In Section 4.1, we present how constraint patterns can be derived from concrete constraints and formally specified as OCL templates and in HOL-OCL. In Section 4.2, we present the concept of an extensible library of constraint patterns. Besides elementary constraint patterns, such a library contains a set of composite constraint patterns, which can be used to combine pattern instances and thus create complex constraints. In Section 4.3, we present an example library of constraint patterns that can be used to express constraints that exclude occurrences of the anti-patterns identified in the previous chapter. We summarize our findings in Section 4.4.

4.1 Deriving and Defining Constraint Patterns

Constraint patterns can be identified by analyzing existing constraints for recurring expressions and abstracting from them. In Section 4.1.1, we use one constraint from Section 3.1 to illustrate how constraint patterns are derived from concrete constraints.

In general, the semantics of constraint patterns can be provided in any language, e. g., parameterized OCL templates as introduced in Section 2.4.1. This has the advantage that constraint patterns can be understood by OCL experts and concrete constraints can be simply instantiated from such templates by providing values for the pattern parameters. In Section 4.1.2, we show how the semantics of constraint patterns can be specified as functions in HOL-OCL. A formalization in HOL-OCL paves the way for defining *composite* constraint patterns, which we cannot express in terms of plain OCL. Furthermore, such a formalization enables the use of constraint patterns for consistency analysis, which will be explained in Chapter 5.

4.1.1 Deriving Constraint Patterns.

In this subsection, we show how constraint patterns can be derived from concrete constraints by abstraction. To this end, we use a constraint that we defined to prevent occurrences of the *Unrestricted Attribute Values* anti-pattern in the previous chapter. We analyze this constraint by investigating those parts of it that can be re-used in other contexts. Subsequently, we replace these parts by variables and define a template that denotes the elicited constraint pattern.

The *Unrestricted Attribute Values* anti-pattern causes that attributes of two or more classes that have some dependency in the system modeled to be unrelated in the corresponding class model. In our company model, the salaries of managers and employees are not related. In Section 3.1.5, we showed a model state in which an employee has a higher salary than his manager. Since we consider this as undesired, we added the constraint *higherSalary* to exclude such instances from the set of valid instances, which we defined as follows.

```
context Manager
inv higherSalary: self.employs−>forAll( e | e.salary < self.salary )
```

Abstractly, this constraint compares (<) an attribute of the context class (salary) to a certain attribute (e.salary) of related objects (self.employs). From this constraint, we thus derive the *Attribute Relation* pattern. Using this pattern, an attribute *contextAttribute* can be related to a *remoteAttribute* by an *operator*. The class containing the *contextAttribute* and the class containing the *remoteAttribute* are related by a *navigation*. We define this pattern as an OCL template as follows.

```
pattern AttributeRelation (navigation:Sequence(Property), remoteAttribute:Property,
                                    operator: OclExpression, contextAttribute:Property) =
self.navigation−>forAll( x | x.remoteAttribute operator contextAttribute)
```

Using this constraint pattern, we can express the constraint *higherSalary* in a more compact way. The definition of this constraint using above pattern reads as follows.

```
context Manager
inv higherSalary: AttributeRelation (employs,salary,<,salary)
```

In Chapter 7, we will present an implementation of our approach in the MDE tool IBM Rational Software Architect (RSA). In this tool, pattern instances are represented in a graphical way in the form of UML collaborations. As can be seen in Figure 4.1, which shows a pattern representation of the constraint *higherSalary*, these collaborations are stereotyped ("Pattern Instance"), have a name ("higherSalary"), indicate the pattern to which this instance belongs ("AttributeRelation"), and provide the parameters of the respective pattern with their types, indicated by the icons next to the parameter names. In addition to the parameters of the respective pattern, an additional parameter *context* needs to be specified as an additional parameter in contrast to the textual pattern representation as shown above in which the context is specified in the header of the OCL invariant.

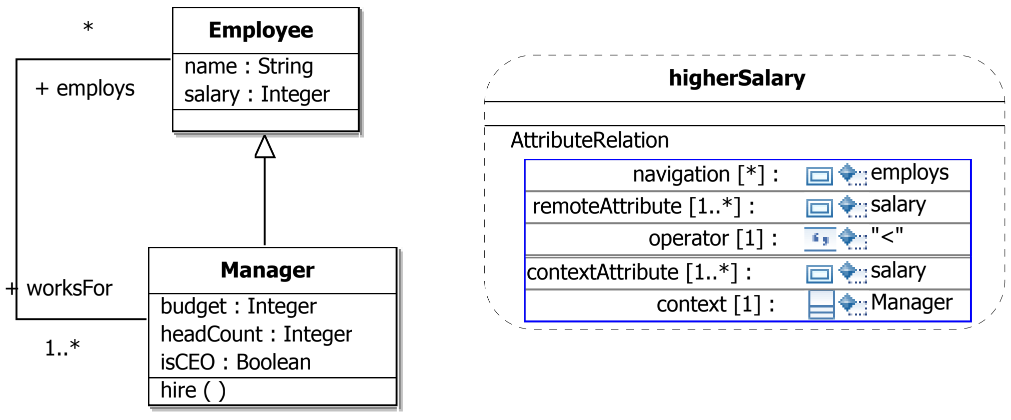


Figure 4.1: Constraint *higherSalary* as represented in RSA.

4.1.2 Representing Patterns in HOL-OCL.

In this section, we explain how constraint patterns can be expressed in terms of HOL-OCL. Such a formalization has two advantages. First, it allows us to define composable constraint patterns which is not possible using OCL templates as introduced before. Second, the HOL-OCL definitions can be used in consistency proofs with the aim of increasing the degree of proof automation. We will use HOL-OCL for proving consistency of UML/OCL models in Section 5.2.

In HOL-OCL, a class invariant is a function that maps an object to an OCL truth value. Thus, a constraint pattern is a function that maps a set of parameter values to a class invariant. In the following definition, we use the type variable 'a to denote an arbitrary set of parameters, which are mapped to an invariant, i. e., a function that maps an object of class 'b to a truth value.

```
types constraintPattern = "'a ⇒
                          ((' τ ,'b::bot)VAL ⇒ 'τ Boolean)"
```

In the following, we show by the example of the *Attribute Relation* pattern how the semantics of constraint patterns can be defined in HOL-OCL. We first define the signature of the constraint pattern. The first parameter is a sequence of Property. In HOL-OCL, this corresponds to a function that maps an object of some class 'a to a collection of objects of some class 'c.

The second parameter denotes an attribute of class 'c. We thus model this parameter as a function from an object of type 'c to a value of type 'b. The third parameter, operator, denotes a binary operator that maps two objects of type 'b to an OCL truth value. The last attribute denotes an attribute of the context class and is thus modeled as a function from 'a to 'b. These parameters are mapped to a constraint, which is a function from class 'a to an OCL truth value.

```
consts
  AttributeRelation  ::
   "(('τ,'a::bot)VAL ⇒ ('τ,'c::bot)Set) ⇒                   −− navigation
    (('τ,'c::bot)VAL ⇒  ('τ,'b::bot)VAL) ⇒                  −− remoteAttribute
    (('τ,'b::bot)VAL ⇒ ('τ,'b::bot)VAL ⇒'τ Boolean) ⇒   −− operator
    (('τ,'a::bot)VAL ⇒ ('τ,'b::bot)VAL) ⇒                   −− contextAttribute
    (('τ,'a::bot)VAL ⇒'τ Boolean)"
```

We define *Attribute Relation* as a lambda expression with one parameter self. Since functions are usually used with a prefix notation in Isabelle/HOL, the OCL expression self.navigation becomes navigation self in HOL-OCL and the remaining pattern parameters are used analogously.

```
defs
  AttributeRelation_def :
   "AttributeRelation navigation remoteAttribute operator contextAttribute ==
   λself. (navigation self)−>forAll(y | operator (remoteAttribute y) (contextAttribute self))"
```

4.2 Composable Constraint Patterns

Although the constraint pattern approach as it has been previously introduced [Ackermann and Turowski, 2006, Costal et al., 2006, Miliauskaitė and Nemuraitė, 2005] reduces both the development time and error rate for model constraints, it has one important restriction. As each pattern represents a subset of all possible constraint expressions, there will be many constraints that are not expressible in terms of existing constraint patterns. This holds even when an extensive pattern library is used.

Therefore, we introduce the notion of *composable constraint patterns* that increases the expressiveness of the constraint pattern approach by allowing to create complex constraints by composing various pattern instances. We divide constraint patterns into *elementary* and *composite* patterns. The set of elementary patterns represents recurring restrictions that have been identified in the literature, and it is extensible by constraint experts. The composite patterns are recursively constructed from elementary patterns and represent logical connectives and quantification, which allows users to create complex constraints from existing patterns.

We relate the constraint patterns using generalization associations. This creates a *taxonomy* of patterns. Such taxonomies as shown in Figure 4.2 give structure to a set of patterns and helps model developers find the right pattern for a specific purpose.

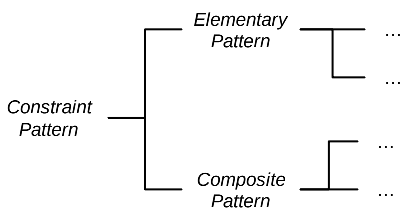


Figure 4.2: Foundations of the constraint pattern taxonomy.

The core of our approach is the class ConstraintPattern, which is a generalization of ElementaryPattern and CompositePattern. Since we have already shown an example of an elementary pattern, we now present an example of a composite pattern.

The *Negation* pattern can be used to logically negate other pattern instances. Syntactically, it has one parameter, which is the constraint to be negated. In the definition of the *Negation* pattern, we exploit that HOL-OCL supports functions as parameters for functions. Thus, we define *Negation* as a function from constraint ((('τ, 'a::bot)Set ⇒ 'τ Boolean)) to constraint.

```
constdefs
  Negation :: "  (('τ, 'a::bot)Set ⇒ 'τ Boolean) ⇒
                 (('τ, 'a::bot)Set ⇒ 'τ Boolean)"
  "Negation P == (λself. not (P self))"
```

Having set the framework for composable constraint patterns, we provide an example library of elementary and composite constraint patterns in the subsequent section. We further show how these constraint patterns can be used to prevent occurrences of the anti-patterns introduced in Chapter 3.

4.3 An Extensible Library of Constraint Patterns

In this section, we present a library of composable constraint patterns. To this end, we first present a collection of constraint patterns in Section 4.3.1 that are motivated by the anti-patterns introduced in Chapter 3. In particular, we introduce for each anti-pattern one or more constraint patterns that can be used to prevent the respective anti-pattern from occurring. Subsequently, we present a set of composite constraint patterns in Section 4.3.2 that can be used to create complex constraints from pattern instances.

4.3.1 Elementary Constraint Patterns.

In this subsection, we present an extensible library of elementary constraint patterns. The idea of elementary constraint patterns is to identify a relevant set of elementary constraints that covers frequently occurring restrictions on a model. The constraint patterns presented in this section can be used to remedy the anti-patterns introduced in Chapter 3. Therefore, we group the patterns introduced in this section according to their respective anti-pattern.

For each pattern, we provide an example constraint from which we derived the pattern. We first define the semantics of the pattern *informally* in English. Subsequently, we *formally* define the semantics of the patterns. Although it would be sufficient to define the semantics in HOL-OCL because corresponding OCL templates could be generated from the HOL-OCL definitions, we also provide equivalent OCL templates because the may be easier to read for model developers.

4.3.1.1 Unrestricted Multiplicity Elements.

In Section 3.1.1, we showed that unbounded multiplicities (*) for associations are on the one hand unavoidable in class models, and on the other hand, they are often a source of low maturity. In this subsection, we present patterns that allow model developers to restrict the cardinality of unbounded associations depending on the context, i. e., attribute values of objects in the instance.

Multiplicity Restriction. In our company model, we modeled that a manager can employ an arbitrary number of employees. However, we required that the number of employees of a manager m depends on the value of the attribute headCount of m. Therefore, we defined the following OCL constraint.

```
context Manager
inv headCountRestriction: self.employs−>size() <= self.headCount
```

This constraint can be represented as an instance of the *Multiplicity Restriction* pattern. The *Multiplicity Restriction* pattern restricts the multiplicity of associations. While the multiplicity of associations can be restricted in a UML class model, this pattern allows model developers to define multiplicity restrictions that depend on properties of the model instance, e. g., an attribute value. We define the pattern as an OCL template as follows.

```
pattern  MultiplicityRestriction (navigation: Sequence(Property),
                                   operator: OclExpression, value:OclExpression) =
 self.navigation−>asSet()−>size() operator value
```

This pattern has three parameters: *navigation*, which is a sequence of properties, thus allowing for the use of OCL navigation expressions such as self.employs.office, *operator*, and *value*, which can be arbitrary OCL expressions. *value* can the name of an attribute or an arbitrary OCL expression. Since self.navigation can evaluate to a multi-set, we use the OCL operator asSet() to cast the resulting collection into a set. In HOL-OCL, the definition reads as follows.

```
constdefs
   MultiplicityRestriction   ::  "  ((' τ,'a::bot)VAL ⇒ (' τ,'c:: collection )VAL ) ⇒
                                    (' τ Integer ⇒ ' τ Integer ⇒ ' τ Boolean) ⇒
                                     ' τ Integer ⇒ ((' τ,'a)VAL ⇒ ' τ Boolean)"

  " MultiplicityRestriction   association operator term == λ self .
     (operator ((association self)−>asSet()::(' τ,'c:: collection )VAL)−>size() term) and
     (0 ≤term)"
```

Using the *Multiplicity Restriction* pattern, we can define the constraint *headCountRestriction* as follows.

```
context Manager
  inv headCountRestriction: MultiplicityRestriction (employs, <=, headCount)
```

Injective Association, Surjective Association, Bijective Association. In Section 3.1.1, we showed that it is generally possible to define associations in class models as injective, surjective, or bijective. However, if the semantics of an association depends on the context of the model instance, e. g., on attribute values, the semantics must be specified with an OCL constraint. The following constraint patterns can be instantiated to specify injectivity, surjectivity, and bijectivity.

```
pattern InjectiveAssociation (property:Sequence(Property)) =
  self.property−>size() = 1 and
  self.allInstances()−>forAll (x,y | x.property = y.property implies x=y)

pattern SurjectiveAssociation(property:Sequence(Property)) =
  self.property.allInstances()−>forAll ( y |
     self.allInstances()−>exists( x | x.property−>includes(y)))

pattern BijectiveAssociation (property:Sequence(Property)) =
  InjectiveAssociation (property) and
  SurjectiveAssociation(property)
```

In HOL-OCL, the definitions read as follows.

```
constdefs
  InjectiveAssociation  ::  " ((' σ U St,'a  ::bot)VAL ⇒
                            (' σ U St,'b  :: collection )VAL) ⇒
                           ((' σ U St,'a  ::bot)VAL ⇒ (' σ U St)  Boolean)"

  " InjectiveAssociation  property ==
   (λself . (property  self)−>size() =1 and
                    ((OclAllInstances  self)−>forAll(x  |
                       ((OclAllInstances  self)−>forAll(y  |
                        ((property  x)  = (property  y)  implies  x  =y )))))) "

constdefs
  SurjectiveAssociation  ::  " ((' σ U St,'a  ::bot)VAL ⇒
                            (' σ U St,'b  ::bot)Set)  ⇒
                           ((' σ U St,'a  ::bot)VAL ⇒ (' σ U St)  Boolean)"

  "SurjectiveAssociation  property ==
   (λself . ∀y ∈ (OclAllInstances  ((property  self )))  ·
           (∃x ∈ (OclAllInstances  self )  ·  ((property  x)−>
           includes((y ::(' σ U St,'b  ::bot)VAL)) ))) "

constdefs
 BijectiveAssociation  ::  " ((' σ U St,'a  ::bot)VAL ⇒
                             (' σ U St,'b  ::bot)Set)  ⇒
                             ((' σ U St,'a  ::bot)VAL ⇒ (' σ U St)  Boolean)"

 " BijectiveAssociation  property ==
  (λself .  InjectiveAssociation  property  self  and SurjectiveAssociation property  self ) "
```

Using these patterns, we can express the constraint *hasManager* from Section 3.1.1 as follows.

```
context Manager inv hasManager:
 if  (not  self .isCEO)
then SurjectiveAssociation(Sequence{employs})
else  true
endif
```

If a manager is not the CEO, the employs association must be surjective, i. e., the manager needs to work for another manager. As an example, consider the constraint that no two employees may work in the same office. This can be expressed through the following pattern instance.

```
context Employee
inv:  InjectiveAssociation ( office )
```

Injectivity and related properties such as surjectivity and bijectivity can also be expressed using multiplicities in the class diagram. However, these patterns become important if an association is restricted under certain *assumptions* only, and not globally for all instances of a model. This can be modeled using a composite pattern (cf. Section 4.3.2).

Object in Collection. Another example constraint that is related to the *Unrestricted Multiplicity Elements* anti-pattern is the following. A company may have the requirement that managers need to work in the same office with at least one of their employees. We specify this constraint in OCL as follows.

```
context Manager inv cooperation:
  self.employs.worksIn.inhabitant−>includes(self)
```

The *Object In Collection* pattern can be used to express that the context element or parts of it must be in a collection of related objects. It has two parameters. The first parameter set denotes a navigation to a set of related elements. The second parameter element denotes the part of the context object that is required to be in set, for example, a certain attribute. If the context object as a whole is required to be in the set, then element must be empty.

```
pattern ObjectInCollection(set:Sequence(Property), element:Sequence(Property)) =
  self.set−>includes(self.element)
```

This OCL template is not correct: If element is empty, the resulting OCL expression contains the term (self.), which is not a valid OCL term. Therefore, we must define the pattern as follows.

```
pattern ObjectInCollection(set:Sequence(Property), element:Sequence(Property)) =
   if  element−>notEmpty()
  then self.set−>includes(self.element)
  else self.set−>includes(self)
  endif
```

HOL-OCL allows us to precisely define this constraint pattern without using conditional statements. In the following definition, element is represented as a function. If element is empty, the identity function λx.x can be used as the according representation in HOL-OCL.

```
constdefs
  ObjectInCollection  ::  "('τ,'b::bot)Set ⇒
                          (('τ,'a::bot)VAL ⇒ ('τ,'b::bot)VAL) ⇒
                          (('τ,'a::bot)VAL ⇒'τ Boolean)"

 ObjectInCollection_def  :
  "ObjectInCollection myset element == λself. (myset−>includes(element self))"
```

We use this constraint pattern to express that a manager needs to work in the same office with at least one employee, using the following instantiation.

```
context Manager inv cooperation:
  ObjectInCollection(employs.worksIn.inhabitant,Sequence{})
```

In a second example, we require that managers must have an employee who has the same salary.

```
context Manager inv fairSalary:
  ObjectInCollection(employs.salary,self.salary)
```

4.3.1.2 Insufficiently Typed Associations.

In Section 3.1.2, we showed that associations between superclasses are often too coarse-grained. In this subsection, we present constraint patterns that allow model developers to specify details about such associations in a concise way.

Type Restriction. In Section 3.1.2, we showed that properties that have a general type, e. g., the property worksIn of type Office, often require further specification, which is not possible in terms of the class-model meta-model. Therefore, OCL constraints need to be defined that restrict the type of properties to a subset of the possible subtypes.

In our example, we want to constrain that employees may not work in single offices and thus defined the following OCL constraint.

```
context Single
  inv onlyManagers: self.inhabitant−>forAll(x | x.oclIsTypeOf(Manager))
```

The *Type Restriction* pattern can be used to restrict an association a that is defined between the context class and a superclass C_0. Using this pattern, it can be enforced that only instances of certain subclasses $C_1, \ldots, C_n$ of C_0, the allowedClasses, may participate in the relation. We use two nested quantifiers for the definition of this pattern, a universal quantifier over the elements in the specified relation and an existential quantifier over the set of allowed classes.

```
pattern TypeRestriction(property:Property, allowedClasses:Set(Class)) =
  self.property−>forAll(x | allowedClasses−>exists(t | x.oclIsTypeOf(t)))
```

In HOL-OCL, the definition reads as follows.

```
constdefs
  TypeRestriction ::
  "(('τ,'a::bot)VAL ⇒ ('τ,'b::bot)Set) ⇒
   ('τ,'c:: collection )Set ⇒
   (('τ,'a::bot)VAL ⇒ 'τ Boolean)"

  TypeRestriction_def :
  "TypeRestriction property allowedClasses == λself.
     (∀ p ∈ ((property self )::(' τ,'b::bot)Set) · (
      ∃ t ∈ (allowedClasses) · ((p ::(' τ,'b::bot)VAL)
                                      −>oclIsTypeOf((t::('τ,'c)VAL)))))"
```

We use this constraint pattern for defining invariant *onlyManagers*, which ensures that only managers can work in single offices. The invariant reads as follows.

```
context Single inv onlyManagers:
  TypeRestriction(inhabitant,Set{Manager})
```

Type Relation. We introduce another constraint pattern that helps refine insufficiently typed associations. The *Type Relation* pattern can be used to restrict an association a that is defined between the context class and a superclass C_0. Using this pattern, it can be enforced that instances of certain subclasses $C_1, \ldots, C_n$ of C_0, the requiredClasses, must participate in the relation.

```
pattern TypeRelation(property:Sequence(Property), requiredClasses:Set(Class)) =
  requiredClasses−>forAll(c | self.property−>exists(p | p.oclIsTypeOf(p)))
```

In HOL-OCL, the definition reads as follows.

```
constdefs
  TypeRelation ::
  "(('τ,'a::bot)VAL ⇒ ('τ,'b::bot)Set) ⇒
   ('τ,'c:: collection )Set ⇒
   (('τ,'a::bot)VAL ⇒ 'τ Boolean)"
  TypeRelation_def :
  "TypeRelation property requiredClasses == λself.
     (∀ t ∈ requiredClasses ·
      (∃ p ∈ ((property self )::(' τ ,'b::bot)Set) ·
              ((p ::(' τ ,'b::bot)VAL)−>oclIsTypeOf((t::('τ,'c)VAL)))))"
```

Whereas we used the previous pattern, *Type Restriction*, for requiring that only managers may work in single offices, this constraint still permits managers to work in cubicle offices. Let us assume that this is undesired and therefore, we define the following constraint that requires managers to work in single offices.

```
context Manager inv onlySingleOffices:
  TypeRelation(office,Set{Single})
```

4.3.1.3 Unrestricted Reflexive Associations.

In Section 3.1.3, we showed that unconstrained reflexive associations allow for instantiations that may be undesired. In particular, instances of reflexive associations can be cyclic, arbitrarily long, or multiple paths, i. e., diamonds, between two objects can exist. In this section, we present three patterns that can be instantiated to avoid such undesired instances.

No Cyclic Dependency. We showed that reflexive associations permit model instances with cyclic paths. In our example, a manager anna worked for berta, who herself worked for cindy, who herself worked for anna. In order to exclude such cycles, we defined the following constraint that ensures that a manager does not appear in the transitive closure of her worksFor association.

```
context Manager

def: closureWorksFor(S:Set(Manager)) : Set(Manager) =
     worksFor−>union((worksFor − S)−>
       collect (m : Manager | m.closureWorksFor(S−>including(self)))−>asSet())

inv noCycles: self.closureWorksFor(Set{})−>excludes(self)
```

To avoid writing such complicated recursive functions, we use the pattern *No Cyclic Dependency*, which uses another pattern *closure* that contains a definition for the transitive closure.

```
pattern NoCyclicDependency(property: Sequence(Property)) =
 self.closure(property)−>excludes(self)

pattern closure(property: Sequence(Property)) =
  self.property−>union(self.property.closure(property))
```

In HOL-OCL, the definition reads as follows.

```
constdefs
  NoCyclicDependency :: "(('a,'b)VAL ⇒ ('a,'b)Set)  ⇒
                        ((' a,'b::bot)VAL ⇒ 'a Boolean)"
```

```
NoCyclicDependency_def :
"NoCyclicDependency property == λself.
  (( transitiveClosure  (makeTuple2(self,property,OclMtSet)))::('a,'b::bot)Set)->
  excludes((self ::('a,'b::bot)VAL)) "
```

For above definition, we use an auxiliary function makeTuple2 that maps a set of parameters to the OCL tuple type. In addition, we use an operator to compute the transitive closure of a reflexive association. Since OCL does not provide such an operator [Object Management Group (OMG), 2003], we need to define it in HOL-OCL. In the following, we define a recursive function transitiveClosure. In HOL-OCL, a measure function must be specified for μ-recursive functions, which we call transitiveClosureMeasure.

```
constdefs
  transitiveClosureMeasure::
  "('a,'b::bot,(('b⇒('b)Set_0),(('b)Set_0)) Tuple_0) Tuple
     ⇒  ('a,'b,(('b⇒('b)Set_0),(('b)Set_0)) Tuple_0) Tuple
     ⇒  'a Boolean"

  transitiveClosureMeasure_def:
  "transitiveClosureMeasure X Y ==
             (let (S1 ::('a,'b::bot)Set) = OclSnd (OclSnd X) in
              let (S2 ::('a,'b)Set) = OclSnd (OclSnd Y) in
              S2->size() ≥S1->size())"


constdefs
   transitiveClosure  ::
  "('a,'b::bot,(('b⇒('b)Set_0),('b)Set_0) Tuple_0) Tuple
       ⇒  ('a,'b)Set"

   transitiveClosure_def :
  "transitiveClosure  args == OclWfrec transitiveClosureMeasure
     (λ f  X.  let  (self ::('a,'b::bot)VAL) = OclFst X in
               let  (S ::('a,'b)Set) = OclSnd (OclSnd X) in
               let  (path ::(('a,'b)VAL⇒('a,'b)Set)) = fixContext (OclFst (OclSnd X)) in
               (path self) ∪ (((path self)-S)->
               collect (m| f  (makeTuple2(m,path,(S ∪(makeSet1 self))))))
     ) args"
```

Using this pattern, we can express the constraint *noCycles* from Section 3.1.3 in a concise way as follows.

```
context Manager
inv noCycles: NoCyclicDependency(worksFor)
```

Path Depth Restriction. Unconstrained reflexive associations make it possible to create instances with arbitrarily long paths. In Section 3.1.3, we showed a path of length seven between two managers anna and helen. We added the following OCL constraint to the model to exclude such instances and restrict the maximum management hierarchy to 5.

```
context Manager

inv smallHierarchy: self.pathDepthWorksFor(4,0)
```

```
def: pathDepthWorksFor(max: Integer, counter: Integer): Boolean =
  if  (counter > max or counter < 0 or max < 0) then false
 else  if  (self.worksFor−>isEmpty()) then true
       else  self.worksFor−>forAll(m:Manager|m.pathDepthWorksFor(max, counter+1))
       endif
 endif
```

We generalize this constraint to the *Path Depth Restriction* pattern, which can be used to limit the maximum length of reflexive associations. To instantiate the pattern, a parameter *property* specifying the association and a parameter *maxDepth* specifying the maximum path depth need to be specified. This pattern uses an auxiliary pattern *pathDepthSatisfied*, which contains the recursive expression.

```
pattern PathDepthRestriction(property: Sequence(Property), maxDepth:Integer) =
  self.pathDepthSatisfied(property,maxDepth−1,0)

pattern pathDepthSatisfied(property: Sequence(Property), max:Integer, counter:Integer) =
  if  (counter > max or counter < 0 or max < 0) then false
 else  if  (self.property−>isEmpty()) then true
       else  self.property−>forAll(m|m.pathDepthSatisfied(property, max, counter+1))
       endif
 endif
```

In HOL-OCL, the definition reads as follows.

```
consts
  PathDepthRestriction ::
  " ((' τ ,' a ::bot)VAL ⇒ (' τ ,' b ::bot)Set) ⇒
    ' τ Integer   ⇒
   ((' τ ,' a ::bot)VAL ⇒  ' τ Boolean)"

defs
  PathDepthRestriction_def:
  "PathDepthRestriction property maxDepth == λself.
   (pathDepthSatisfied (mkOclTuple 0 (mkOclTuple (maxDepth − 1)
                                        (mkOclFnTuple self property))))"
```

Analogously to the transitive closure, we define a recursive function pathDepthSatisfied, which we use in the definition of the *Path Depth Restriction* pattern. The definition of pathDepthSatisfied reads as follows.

```
consts
  pathDepthSatisfied ::
  " (' a ::bot,Integer_0 ,( Integer_0 ,(' b ::bot ,(' b ::bot⇒(' b ::bot)Set_0)) Tuple_0) Tuple_0) Tuple
    ⇒ ' a  Boolean"

defs
  pathDepthSatisfied_def :
  "pathDepthSatisfied args == OclWfrec pathDepthSatisfiedMeasure
    (λ f X. let (counter ::' b ::bot Integer) = OclFst X in
            let (MAX::'b Integer) = OclFst (OclSnd X) in
            let ( self ::(' b ,' a ::bot)VAL) = (OclFst (OclSnd (OclSnd X))) in
            let (path ::((' b ,' a ::bot)VAL⇒('b,'a)Set)) =
                 fixContext (OclSnd (OclSnd (OclSnd X))) in
             if (counter >MAX or counter <0 or MAX <0)
            then OclFalse
            else  if  (OclIsEmpty (path self))
                 then OclTrue
                 else (∀ z ∈ ((path  self  )::(' b,' a)Set)  ·
                        (f  (mkOclTuple (counter+1)
                        (mkOclTuple MAX (mkOclFnTuple (z::('b,'a)VAL) path)))))
                 endif
            endif
   )  args"
```

The definition of the measure function that ensures termination reads as follows.

```
consts pathDepthSatisfiedMeasure ::
" (' a::bot,Integer_0 ,( Integer_0 ,(' b::bot ,(' b::bot⇒ ('b::bot)Set_0))  Tuple_0)  Tuple_0)  Tuple ⇒
 (' a::bot,Integer_0 ,( Integer_0 ,(' b::bot ,(' b::bot⇒ ('b::bot)Set_0))  Tuple_0)  Tuple_0)  Tuple ⇒
                                                         'a   Boolean"

defs
  pathDepthSatisfiedMeasure_def:
  "pathDepthSatisfiedMeasure == (λ X Y.
     let  (counter1 ::'b::bot  Integer)  = OclFst X in
     let  (counter2 ::'b::bot  Integer)  = OclFst Y in
     counter2 >counter1)"
```

We use this pattern now to restrict the number of hierarchy levels in our company model to five. Using the pattern, this constraint can be defined as follows.

```
context Manager
inv smallHierarchy: PathDepthRestriction(worksFor,5)
```

Unique Path. We have seen that so-called diamond configurations (cf. Definition 14) can occur in object graphs when reflexive associations are unconstrained. In our example in Section 3.1.3, anna managed two other managers, berta and cindy, who share one employee, daniela. Such configurations can cause problems because there is more than one path between anna and daniela. We excluded such instances with the constraint *noDiamond*.

```
context Manager inv noDiamond:
  self.worksFor−>exists(m1,m2 | m1−>closure(worksFor)−>intersect(
                                   m2−>closure(worksFor))−>notEmpty()
                          implies  m1=m2)
```

We generalize from this concrete constraint and derive the *Unique Path* pattern. This pattern allows one to easily exclude diamond-shaped instances for its parameter, *property*. The definition of the pattern reads as follows.

```
pattern UniquePath(property: Sequence(Property)) =
  self.property−>exists(m1,m2 | m1−>closure(property)−>intersect(
                                   m2−>closure(property))−>notEmpty()
                             implies  m1=m2)
```

In HOL-OCL, the definition reads as follows.

```
consts
  UniquePath :: " ((' τ,'a::bot)VAL ⇒ (' τ,'b::bot)Set)  ⇒
                  ((' τ,'a::bot)VAL ⇒ 'τ Boolean)"
```

```
defs
 UniquePath_def :
  "UniquePath property == λself.
     (∃ m1 ∈((property self )::(' τ,'b::bot)Set) · (∃ m2 ∈(property self ) ·
      (((( transitiveClosure (makeTuple2(m1,property,OclMtSet))::(('τ,'b::bot)Set)) ∩
           transitiveClosure (makeTuple2(m2,property,OclMtSet)))−>isEmpty())
            implies ((m1::('τ,'b::bot)VAL) =(m2::('τ,'b::bot)VAL)))))"
```

A well-known example configuration that can be excluded with this pattern was identified in [Reiter and Criscuolo, 1981] and became famous by the rather dramatic name of "diamond of death" in object-oriented programming languages. In this configuration, four classes A, B, C and D are in the generalization relation $\prec = \{(A, B), (A, C), (B, D), (C, D)\}$. If B and C inherit a structural feature x from A, it is unclear whether D inherits $B :: x$ or $C :: x$. Thus, the path from a class to each superclass of its superclasses should be unique.

```
context Class
inv: UniquePath(superClass.superClass)
```

4.3.1.4 Missing Unique Identification.

Unique Identifier. We showed in Section 3.1.4 that objects should be uniquely identifiable. We used the following constraint to express that each employee can be uniquely identified by the name and by the office that the employee inhabits.

```
context Employee
inv uniqueness: self.allInstances()−>isUnique(e|Tuple(x=e.name,y=e.worksIn))
```

From this constraint, we derive the *Unique Identifier* pattern. This pattern is also known as "semantic key" [Ackermann and Turowski, 2006], "primary identifier" [Miliauskaitė and Nemuraitė, 2005] or "identifier" [Costal et al., 2006] pattern in the literature. The *Unique Identifier* pattern has one parameter *property*, which denotes a tuple of properties that have to be unique for each object of the context class.

```
pattern UniqueIdentifier(property:Tuple(Property)) =
  self.allInstances()−>isUnique(property)
```

In HOL-OCL, the definition reads as follows.

```
consts
  UniqueIdentifier ::
    " ((' τ,'a::bot)VAL ⇒ (' τ,'b::bot)VAL)⇒ ((' τ,'a::bot)VAL ⇒ ' τ Boolean)"

defs
  UniqueIdentifier_def :
  "UniqueIdentifier accessor == λself.
     (OclAllInstances self)−>forAll(y | ((OclAllInstances self)−>forAll(z |
       ((¬(y=z)) implies ¬(accessor y =accessor z)))) )"
```

The constraint *uniqueness*, which requires that instances of the Employee class are uniquely identifiable by their name and office, can be expressed using the *Unique Identifier* pattern as follows.

```
context Employee
inv: UniqueIdentifier(Tuple{x1=name, x2=worksIn})
```

4.3.1.5 Unrestricted Attribute Values.

In Section 3.1.5, we showed that textual constraints are necessary to express relations between properties. In the following, we introduce three constraint patterns that can be used to remedy the *Unrestricted Attribute Values* anti-pattern.

Attribute Sum Restriction. In Section 3.1.5, we showed a different source of low maturity in which the cardinality of the association depends on the relation of several attributes. In this example, the number of employees that managers can employ depends on the budget of the respective manager and the salaries of his employees. The OCL constraint that expresses this dependency reads as follows.

```
context Manager
inv budgetRestriction: self.employs.salary−>sum() <= self.budget
```

To capture this constraint, we use the *Attribute Sum Restriction* pattern, which has three parameters. Besides the parameter *navigation*, which denotes a path expression to a related class, this pattern has two parameters. Parameter *summation* refers to the property in the context class that denotes the value that must not be exceeded, and *summand* refers to the property in the related class that is accumulated.

```
pattern AttributeSumRestriction(navigation: Sequence(Property),
                                summand: Property, summation: Property) =
 self.navigation.summand−>sum() <= summation
```

In HOL-OCL, the definition reads as follows.

```
constdefs
  AttributeSumRestriction :: "(('τ,'a::bot)VAL ⇒'τ Integer) ⇒
                              (('τ,'a::bot)VAL ⇒ ('τ,'b::bot)Set) ⇒
                              (('τ,'b::bot)VAL ⇒'τ Integer) ⇒
                              (('τ,'a::bot)VAL ⇒'τ Boolean)"

  "AttributeSumRestriction sum nav summand == λself.
     ((OclCollect ((nav self )::('τ,'b::bot)Set) (summand::(('τ,'b::bot)VAL ⇒
                 'τ Integer )))::('τ,'τ Integer)Set)−>sum() ≤(sum self)"
```

Using this constraint pattern, we can express the constraint *budgetRestriction* as follows.

```
context Manager
inv budgetRestriction: AttributeSumRestriction(employs,salary,budget)
```

Attribute Relation. In Section 3.1.5, we showed a state of the company model in which the manager has a lower salary than his employee. We used constraint *higherSalary* to exclude such states, which we defined as follows.

```
context Manager
inv higherSalary: self.employs−>forAll( e | self.salary > e.salary )
```

We abstract from this constraint to the *Attribute Relation* pattern. Using this pattern, an attribute *contextAttribute* can be related to a *remoteAttribute* by an *operator*. The class containing the *contextAttribute* and the class containing the *remoteAttribute* are related by a *navigation*. We define this pattern as an OCL template as follows.

```
pattern AttributeRelation (navigation:Sequence(Property), remoteAttribute:Property,
                                        operator: OclExpression, contextAttribute:Property) =
self.navigation−>forAll( x | x.remoteAttribute operator contextAttribute)
```

We formally define the semantics in HOL-OCL as follows.

```
constdefs
  AttributeRelation ::
   " ((' τ ,'a::bot)VAL ⇒ (' τ ,'c::bot)Set) ⇒                    −− navigation
     ((' τ ,'c::bot)VAL ⇒  (' τ ,'b::bot)VAL) ⇒                   −− remoteAttribute
     ((' τ ,'b::bot)VAL ⇒ (' τ ,'b::bot)VAL ⇒ ' τ Boolean) ⇒  −− operator
     ((' τ ,'a::bot)VAL ⇒ (' τ ,'b::bot)VAL) ⇒                    −− contextAttribute
     ((' τ ,'a::bot)VAL ⇒ ' τ Boolean)"

   "AttributeRelation navigation remoteAttribute operator contextAttribute ==
      λself. (navigation self)−>forAll(y |
         operator (remoteAttribute y) (contextAttribute self)) "
```

Using this pattern, we can express the constraint *higherSalary* as follows.

```
context Manager
inv higherSalary: AttributeRelation (employs,salary,<,salary)
```

Attribute Value Restriction. In the background section on OCL, we introduced the constraint *budgetGreaterZero*, which we defined as follows.

```
context Manager
inv: budgetGreaterZero: self.budget > 0
```

This constraint represents a common kind of constraint, namely simple value restrictions for attributes. We therefore introduce the *Attribute Value Restriction* pattern, which can be used to restrict the values of attributes for all instances of the attributes' class. We define it as OCL template as follows.

```
pattern AttributeValueRestriction (property:Property,operator,value:OclExpression) =
  self.property operator value
```

In HOL-OCL, the definition reads as follows.

```
consts
   AttributeValueRestriction ::
    " ((' τ ,'a::bot)VAL ⇒ (' τ ,'b::bot)VAL) ⇒
      ((' τ ,'b::bot)VAL ⇒ (' τ ,'b::bot)VAL ⇒ ' τ Boolean) ⇒ (' τ ,'b::bot)VAL ⇒
      ((' τ ,'a::bot)VAL ⇒ ' τ Boolean)"

defs
   "AttributeValueRestriction attribute operator term ==
      λself. operator ( attribute self) term"
```

Using the pattern, we can express the constraint budgetGreaterZero as follows.

```
context Manager
inv: AttributeValueRestriction (budget, >=, 0)
```

4.3.1.6 Summary.

With these patterns, we have introduced a reusable remedy for each anti-pattern presented in Section 3.2. As seen, there are often several constraint patterns that can be used to prevent occurrences of the respective anti-pattern. Note that there can be more constraint patterns for each anti-pattern because we elicited them heuristically based on modeling experience. Thus, our collection of constraint patterns is not complete.

Figure 4.3 illustrates all elementary constraint patterns from our library as a taxonomy. Furthermore, this figure also contains the composite constraint patterns that we define in the subsequent section.

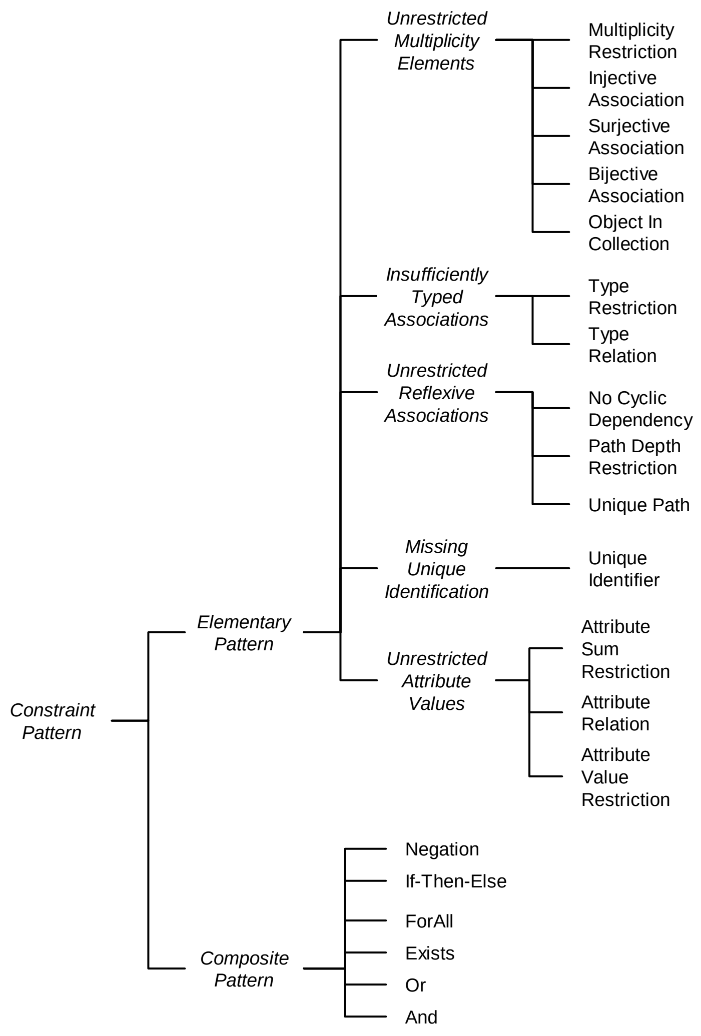


Figure 4.3: Overview of the constraint pattern library.

4.3.2 Composite Constraint Patterns.

Apart from elementary constraint patterns, each of which restrict a basic property of a model, composite constraints can be used to express complex properties by integrating an arbitrary number of other constraints (either elementary or composite). Thus, complex constraints can be developed in a structured way by combining several simple constraints. So far, we have identified the composite patterns *Negation*, *If-Then-Else*, *Exists*, *ForAll*, *Or*, and *And*, which are illustrated in Figure 4.3.

Since composite constraint patterns are higher-order constructs, i. e., they represent constraints over constraints, they cannot be represented by OCL templates as previously used in this thesis. We hence only provide the HOL-OCL definitions for the composite patterns.

In the definitions for our composite patterns in Section 4.3.2, we will use operators for the conjunction and disjunction of arbitrary lists of predicates. Such higher-order operators do not exist in OCL, but we can define them in HOL-OCL as follows.

```
consts
  oclAND :: " ((' τ ,'a::bot)VAL ⇒'τ Boolean)list
              ⇒ ((' τ ,'a::bot)VAL ⇒'τ Boolean)"

primrec
  "oclAND [] = (λx. OclTrue)"
  "oclAND (x#xs) = (λa. ((x a) and ((oclAND xs a))))"
```

```
consts
  oclOR :: " ((' τ ,'a::bot)VAL ⇒'τ Boolean)list ⇒
             ((' τ ,'a::bot)VAL ⇒ 'τ Boolean)"

primrec
  "oclOR [] = (λa. OclFalse)"
  "oclOR (x#xs) = (λa. ((x a) or ((oclOR xs a))))"
```

Negation. The *Negation* pattern can be used to logically negate another pattern instance. We define it in HOL-OCL as a function from constraint ((' τ, 'a::bot)Set ⇒ 'τ Boolean) to constraint.

```
constdefs
  Negation :: " ((' τ, 'a::bot)Set ⇒ 'τ Boolean) ⇒
               ((' τ, 'a::bot)Set ⇒ 'τ Boolean)"
  "Negation P == (λself. not (P self))"
```

As an example constraint, we require that the association worksIn from Employee to Office is not surjective, i. e., there must be at least one empty office. Using the *Negation* and *Surjective Association* pattern, we define this constraint as follows.

```
context Employee
inv: Negation(SurjectiveAssociation(Sequence{worksIn}))
```

If-Then-Else. The *If-Then-Else* pattern denotes an if-then-else expression, which can be used to express logical entailment between pattern instances. If the context element of the constraint satisfies all properties, which are denoted by a set of constraints, it also needs to satisfy all then constraints, otherwise, it needs to satisfy all else constraints. We define the semantics of the *If-Then-Else* pattern as follows.

```
constdefs
  IfThenElse :: " ((' τ ,'a::bot)VAL ⇒'τ Boolean)list ⇒
                 ((' τ ,'a::bot)VAL ⇒'τ Boolean)list ⇒
                 ((' τ ,'a::bot)VAL ⇒'τ Boolean)list ⇒
                 ((' τ ,'a::bot)VAL ⇒'τ Boolean)"
```

```
 "IfThenElse ifClause thenClause elseClause == λself.
   if  (oclAND ifClause self)
     then (oclAND thenClause self)
     else (oclAND elseClause self)
   endif"
```

If no *else* clause is needed, an empty list can be specified as parameter value for elseClause because we defined the conjunction of an empty list as equivalent to *true* (oclAND [] ≡ OclTrue).

In the following example constraint, we require that managers who are the CEO must have a budget of at least one million. Note that the *else* clause is empty, which corresponds to *true*.

```
context Manager
inv: IfThenElse(AttributeValueRestriction(isCEO,=,true),
                AttributeValueRestriction(budget,>,1000000),
               )
```

Note that the *if* and *then* clause must be collections. For better readability, we left out the set constructors as used in the example for the *Negation* pattern.

Exists. The *Exists* pattern is used to express existential quantification. In particular, it can be used to restrict a pattern instance such that it must only hold for at least one object, not for *all* objects of a certain class. We define its semantics in HOL-OCL as follows.

```
constdefs
  Exists :: "('τ, 'a::bot)Set ⇒ (('τ, 'a::bot)VAL ⇒'τ Boolean)list ⇒ 'τ Boolean"
  "Exists S P == S−>exists(y | (oclAND P y))"
```

Using the *Exists* pattern, we state the requirement that every employee must have a manager whose head count is exactly one. The constraint reads as follows.

```
context Employee
inv: Exists(worksFor,
            AttributeValueRestriction(headCount,=,1))
```

For-All. The *ForAll* constraint pattern is used to express universal quantification over a set of objects. We define its semantics in HOL-OCL as follows.

```
constdefs
  ForAll :: "(('τ, 'a::bot)VAL ⇒'τ Boolean)list ⇒ ('τ, 'a::bot)Set ⇒ 'τ Boolean"
  "ForAll P S == S−>forAll(y | (oclAND P y))"
```

Using the *ForAll* pattern, we can state that employees must have a salary of at least 4000 to be allowed to work in an office. This constraint reads as follows.

```
context Office
inv: ForAll(inhabitant,
            AttributeValueRestriction(salary,>=,4000))
```

Or. The *Or* pattern is used to express the disjunction of a set of pattern instances. It uses the previously defined function oclOR and is defined in HOL-OCL as follows.

```
constdefs
  Or :: "((' τ, 'a::bot)Set ⇒ 'τ Boolean)list ⇒
          ((' τ, 'a::bot)Set ⇒ 'τ Boolean)"
  "Or P == oclOR P"
```

Using the *Or* pattern, we can express that an office has desks or it does not have any inhabitants. The constraint reads as follows.

```
context Office
inv: Or(AttributeValueRestriction (desks,>,0),
         MultiplicityRestriction (inhabitant ,=,0))
```

And. The *And* pattern is used to express the conjunction of a set of pattern instances. It uses the previously defined function oclAND and is defined in HOL-OCL as follows.

```
constdefs
  And :: "((' τ, 'a::bot)Set ⇒ 'τ Boolean)list ⇒
          ((' τ, 'a::bot)Set ⇒ 'τ Boolean)"
  "And P == oclAND P"
```

Using the *And* pattern, we can express that the name of each employee must be unique and the budget of each manager must be higher than the employee's salary. The constraint reads as follows.

```
context Employee
inv: And(UniqueIdentifier({name}),
         AttributeRelation (worksFor,budget,>,salary))
```

4.3.3 Using Arbitrary Constraints in Composite Patterns.

In case that not all parts of composite constraints can be expressed using constraint patterns, it is desirable to allow model developers to use arbitrary OCL expressions as parameter values in instances of composite patterns. Therefore, we define the *Literal OCL* pattern, which wraps arbitrary OCL expressions such that they can be used in instances of composite patterns. The definition of this pattern in terms of an OCL template reads as follows.

```
pattern LiteralOCL(expression:OclExpression) =
 expression
```

In HOL-OCL, the definition reads as follows.

```
  LiteralOCL :: "((' τ, 'a::bot)Set ⇒ 'τ Boolean) ⇒
                 ((' τ, 'a::bot)Set ⇒ 'τ Boolean)"
  "LiteralOCL P == (λself. P self)"
```

Obviously, the following two invariants are equivalent because the second invariant wraps the first invariant in an instance of the *Literal OCL* pattern.

```
context Office
inv: self.desks >= 0
inv: LiteralOCL(self.desks >= 0)
```

4.4 Summary

In this chapter, we have introduced the concept of an extensible library of generic constraint patterns. In addition, we have introduced an example library of constraint patterns that provides effective patterns for remedying the anti-patterns presented in Chapter 3. We have added a high degree of expressiveness to existing pattern-based approaches by adding logical structure and classifying patterns into elementary and composite patterns.

Such a flexible pattern-based approach offers an important improvement for the development of constraint specifications compared to previous constraint-pattern approaches and to writing OCL constraints by hand. In contrast to previous approaches that use constraint patterns, our approach offers a larger set of elementary constraint patterns and, more importantly, the concept of composable constraint patterns. This is an improvement in expressiveness compared to previous approaches. In contrast to writing OCL constraints by hand, our approach helps to avoid many syntactic and structural errors because the developer can generate OCL code instead of writing it by hand. Furthermore, using our solution, more concise constraint specifications can be developed, which helps to decrease development time substantially. We evaluate our approach in Chapter 8 where we present case studies and compare the pattern-based specification method to the traditional code-based specification method.

Chapter 5

Consistency of Constraint Specifications

In the previous chapter, we showed how constraint patterns can be used to simplify the development of constraint specifications. However, even when such a pattern approach is used, model developers can still inadvertently develop inconsistent constraint specifications, which makes it impossible to instantiate the constrained model. Inconsistencies need to be detected in the development process before the model is deployed and used, e. g., for code generation.

Existing publications on this topic either discuss consistency of models without OCL constraints, do not provide precise definitions, or merely define individual aspects of consistency [Ahrendt et al., 2005, Berardi et al., 2005, Gogolla et al., 2005, Kaneiwa and Satoh, 2006, Maraee and Balaban, 2007, Queralt and Teniente, 2006]. There is no comprehensive examination of consistency for UML/OCL models or a well-established notion of consistency of OCL specifications. Furthermore, it is unclear what aspects of consistency are relevant in practice. In particular, there is no literature that covers consistency in incremental development approaches. For example, in early development phases, weaker definitions of consistency may be more appropriate than in later development phases.

The contribution of this chapter is to investigate consistency concerns for OCL specifications and their practical relevance for different phases of development. We provide formal definitions for the different concerns, and show how consistency analysis of an OCL specification can be integrated into an MDE process. We investigate in Section 5.1 what aspects of consistency for an OCL specification are important in practice and formally define several consistency notions appropriate for UML/OCL. In Section 5.2, we show an example of how these consistency definitions can be used for a given constraint specification by generating proof obligations and checking them with a theorem prover. We summarize our findings in Section 5.3.

5.1 UML, OCL, and the Notion of Consistency

Before consistency analysis can be performed, we must define the precise meaning of consistency for UML/OCL models. Requiring consistency in the classical sense of FOL for UML/OCL models means that there must exist a state of the model that satisfies all invariants in the OCL constraint specification and that contains at least one object for each class in the UML model. As indicated in Section 2.5.2, UML/OCL has some characteristic features whose role for consistency needs to be investigated. In this section, we therefore discuss subtyping, abstract classes, and state finiteness, and we eventually weaken the classical notion of consistency to make it suitable for UML/OCL.

5.1.1 Notions of Consistency.

5.1.1.1 Subtyping.

The notion of subtype (or *specialization* in UML terminology) is an important concept in object-oriented modeling, as it allows model developers to extend concepts and use specialized concepts to replace more general concepts. An instance of a class C' is also an instance of its superclass C and thus, needs to satisfy the invariants for both C and C'.

The invariants specified for a superclass usually need not be added explicitly to its subclasses. Instead, subclasses are usually annotated with *additional* invariants that their instances must satisfy, as in the following example. The example constraints require that the salary of all employees is positive and, in addition, the budget of all managers must be positive for the company model.

```
context Employee inv positive_salary:
  inv: salary > 0

context Manager inv positive_budget:
  inv: budget > 0
```

In UML, an instance of a specific class is also considered an instance of each of its superclasses [Object Management Group (OMG), 2006c, Sect. 7.3.20]. Thus, the instance of a specific class needs to satisfy the invariants of all its superclasses. We call this property *subtype consistency* and define it as follows:

Definition 15 (Subtype Consistency). *A model M is* subtype-consistent *if and only if for all $C,\ C' \in M$, where C' is a subclass of C, an instance of C' is also a valid instance of C in every state τ, or formally (in HOL-OCL),*

$$\begin{aligned}\forall C, C' \in M.\ \forall \tau.\ \tau \vDash \quad & C'\ \text{::allInstances()}\\ & \rightarrow\text{forall}(x \mid \big(x\ \text{.oclIsTypeOf}(C')\ \text{and}\ x\ \text{.oclIsKindOf}(C)\big)\\ & \text{implies}\ (\text{not}\ x\ \text{.oclAsType}(C)\text{.oclIsUndefined()}))\ .\end{aligned}$$

The notion of *subtype consistency* defined in Definition 15 is popularly known as the *Liskov principle* [Liskov and Wing, 1994]. This principle requires that if ϕ is a property that holds for objects of type T, then ϕ should hold for objects of type S, where S is a subtype of T ($S \leq T$). It is also known as *class subsumption* [Berardi et al., 2005] for class models without OCL invariants or *structural subtyping* [Ahrendt et al., 2005] for models with OCL, which requires that the invariant of a subclass implies the invariant of its superclass.

5.1.1.2 Abstract Classes and Interfaces.

In this subsection, we describe how the notion of subtype consistency can be used to include abstract classes and interfaces in our consistency definitions to come. At first glance, any model that contains abstract classes cannot be consistent because an abstract class(ifier) “*does not provide a complete declaration and can typically not be instantiated.*”

according to the UML specification [Object Management Group (OMG), 2006c]. Interfaces are treated similarly in the UML specification: "*Since interfaces are declarations, they are not instantiable.*"

However, we argue the following for abstract classes and interfaces: Model developers usually create an abstract class or interface C assuming that there should be at least one concrete class C' that specializes C because otherwise, C would be superfluous. Thus, if C' can be instantiated, also C can be instantiated because an instance of a subclass C' is also considered an instance of all of its superclasses in UML. Consequently, the classical notion of consistency *is* applicable to UML with its concepts of interfaces and abstract classes, subtype consistency provided. If subtype consistency was *not* a prerequisite, the following invariant added to the model in Figure 2.5 would erroneously result in a *consistent* model because class Manager, which is also an instance of Employee, could still be instantiated when subtype consistency is not required.

```
context Employee inv contrived:
  false
```

Note that in early phases of development, there may be abstract classes for which no concrete subclasses has been defined yet. Thus, we discuss fine-grained notions of consistency in Section 5.1.1.4.

Without requiring subtype consistency, we can specify a state containing one object o of type Manager, for which o.oclIsKindOf(Employee) and o.oclIsKindOf(Manager) holds. Thus, Employee::allInstances()−>notEmpty() and Manager::allInstances()−>notEmpty(). However, the model is not subtype-consistent because an instance of Manager cannot be an instance of Employee, which is required by Definition 15 for subtype consistency (not o.oclAsType(Employee).oclIsUndefined()).

Surprisingly, we have not found any discussion in the literature about the role of abstract classes and interfaces for the consistency of a UML/OCL model.

5.1.1.3 Finite vs. Infinite States.

The state of a given UML/OCL model can potentially contain an infinite number of objects. Thus, there can be models that are consistent, but no model state exists with a finite number of objects in which all class invariants are satisfied. Since states with an infinitely large number of objects are rarely desirable in practice, we discuss the problem of finite and infinite states in this subsection.

Recall the constraint *noCycles* from Section 4.3.1, which is defined using the *No Cyclic Dependency* pattern and disallows cycles in the management hierarchy of a company, i. e., managers must not be their direct or indirect (via transitivity) manager.

Since the multiplicity constraints in the UML model require that each employee is associated with at least one manager, *noCycles* can only be satisfied by a state in which there are either no employees at all or infinitely many managers. However, states with no employees violate consistency, and states with infinitely many managers are undesirable.

We can require that a model M needs to have at least one finite state by adding the following requirement to M:

$$\forall C \in M.\ \exists \tau.\ \tau \vDash \text{not}\ C\ \text{::allInstances()}\ \rightarrow \text{size()}\ \text{.oclIsUndefined()} \tag{5.1}$$

In this expression, we exploit that HOL-OCL supports infinite sets, e. g., in HOL-OCL the operation ::allInstances() can return an infinite set, and the size() operator is undefined for infinite sets in HOL-OCL (cf. Section 2.6.2). Thus, requiring the existence of a state in which the size of ::allInstances() is defined ensures that the model does not have infinite instances only.

We have found little discussion in the literature about the role of finite and infinite instances for the consistency of a UML/OCL model. Whereas the existence of a finite state is an important requirement [Maraee and Balaban, 2007], most publications on this topic implicitly assume that infinitely large instances are witnesses for the consistency of a model without further discussion.

Having investigated the characteristic features of UML/OCL, we can check whether our example company model is consistent, and it will turn out that the classical notion of consistency is often too strong in UML/OCL, which motivates weaker notions of consistency. In the following subsection, we incrementally weaken the classical notion of consistency and thus obtain fine-grained notions of consistency for UML/OCL.

5.1.1.4 Fine-grained Notions of Consistency.

According to the classical notion of consistency, the company model is consistent, but we have seen in the previous subsection that there is no finite state of the model that satisfies all invariants. However, if we change the multiplicity of the worksFor association end from $1..*$ to $*$, we get a model company'. This model has finite states, of which we show one in Figure 5.1 as τ.

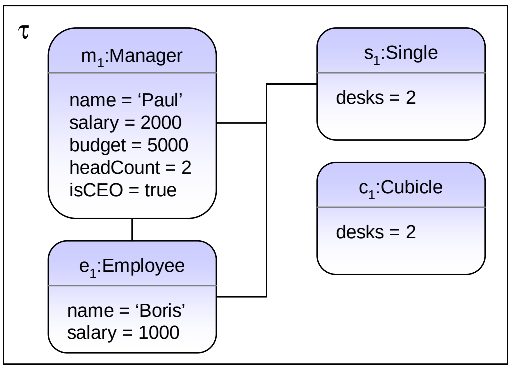


Figure 5.1: Consistent state of the company' model.

However, it is not always possible to find a single state in which all classes of a given model can be instantiated. Consider the following invariant, which allows companies to have either offices of type Single or of type Cubicle, but not both.

```
context Company inv workConditions:
   Single::allInstances()->isEmpty() or
   Cubicle::allInstances()->isEmpty()
```

The company model with workConditions is inconsistent with respect to the classical notion of consistency because no instance of the model exists in which all classes are instantiated *and* the workConditions constraint is satisfied. However, the model developer may have deliberately created this scenario described above and does not consider it inconsistent. To reconcile the developer's intention with a formal notion of consistency, we need to weaken the classical notion of consistency. To this end, we investigate what notions of consistency a model developer can be concerned about.

Can each class be instantiated in the same state? As shown, if a set of constraints is consistent in the classical sense, there exists at least one state in which all classes can be instantiated. This is a strong requirement, but it can be useful for UML/OCL models with strong dependencies between the classes in the model, i. e., an instance of a class can only exist if instances of all other classes also exist.

In the remainder of this chapter, we refer to the classical notion of consistency as *strong consistency*, which we define as follows:

Definition 16 (Strong Consistency). *A UML/OCL model M is* strongly-consistent *if and only if there exists a state in which all classes of M are instantiated, or formally,*

$$\exists\tau.\forall C \in M.\ \tau \vDash C\ \text{::allInstances()} \rightarrow \text{exists}(x \mid x\ \text{.oclIsKindOf}(C)),$$

which is equivalent to

$$\exists\tau.\forall C \in M.\ \tau \vDash C\ \text{::allInstances()} \rightarrow \text{notEmpty()}\ . \tag{5.2}$$

Can each class be instantiated in some state? Since strong consistency is sometimes too strong, we weaken the previous requirement and ask for the existence of a set of states for M such that each class in M is instantiated in at least one of these states.

This notion of consistency is appropriate for the workConditions constraint: This constraint is satisfied by a state τ_1 that contains objects of type Single only and by a state τ_2 that contains objects of type Cubicle only. Thus, each class of the UML model can be instantiated, although in different states, which is visualized in Figure 5.2. We call this weaker notion of consistency *class consistency* and define it as follows:

Definition 17 (Class Consistency). *A UML/OCL model M is* class-consistent *if and only if for each class $C_i \in M$ there exists a state that contains an instance of class C_i, or formally,*

$$\forall C \in M.\ \exists\tau.\ \tau \vDash C\ \text{::allInstances()} \rightarrow \text{notEmpty()}\ . \tag{5.3}$$

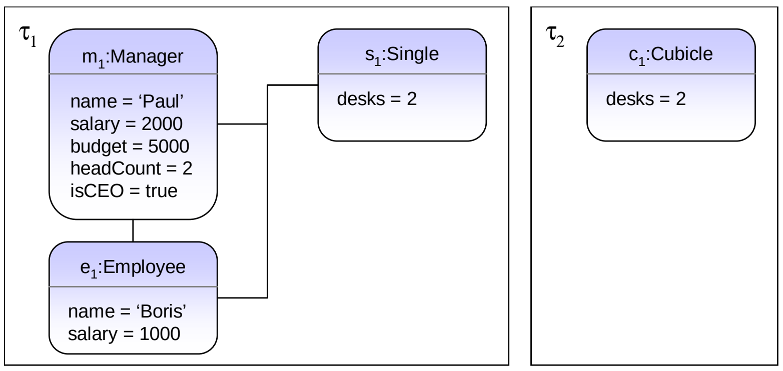


Figure 5.2: Witnesses for the class consistency of company'.

Clearly, every strongly-consistent model is also class-consistent, i. e., strong consistency implies class consistency, but not vice versa.

Can any class of the model be instantiated? In early phases of the software development process, it can happen that certain classes in a model do not yet have an implementation. It can also happen that certain classes are still in the model, but using them is discouraged or they do not have an implementation anymore.

According to the previously defined notions of consistency, such models are neither strongly-consistent nor class-consistent because not all classes can be instantiated. However, as explained above, such a situation may be desired, e. g., in early phases of development. Thus, the developer may want to know whether there is a nonempty subset of all classes in M that can be instantiated. We capture this notion of consistency as *weak consistency*.

Definition 18 (Weak Consistency). *A UML/OCL model M is* weakly-consistent *if and only if a nonempty subset of the classes of M can be instantiated in some state, or formally,*

$$\exists \mathcal{C} \subseteq M.\, \mathcal{C} \neq \emptyset \wedge \forall C \in \mathcal{C}.\, \exists \tau.\, \tau \vDash C\,\texttt{::allInstances()}\rightarrow\texttt{notEmpty()}\,. \tag{5.4}$$

Every class-consistent model is also weakly-consistent, but not vice versa. Given a UML/OCL model that is weakly-consistent, the model developer can incrementally adapt the model to achieve a stronger notion of consistency. If a model M is not even weakly-consistent, we call it *inconsistent*, which we define as follows:

Definition 19 (Inconsistency). *A UML/OCL model M is* inconsistent *if and only if there does not exist a state in which any class of M can be instantiated.*

The three different notions of consistency can be found in the literature, but neither is the difference between the notions used motivated nor are the notions precisely defined. The notion of strong consistency can be found in [Kaneiwa and Satoh, 2006, Maraee and Balaban, 2007], the notions of class consistency and weak consistency can be found in [Berardi et al., 2005, Maraee and Balaban, 2007, Queralt and Teniente, 2006].

5.1.2 Discussion.

We have investigated the role of the characteristic features of UML/OCL for consistency. Consequently, we have presented different consistency definitions and discussed finiteness of model instances. We propose that practically relevant UML/OCL models should be subtype-consistent, class-consistent, and have at least one finite instance for the following reasons: We consider subtype consistency a necessary requirement for object-oriented models [Liskov and Wing, 1994], and if a model is not subtype-consistent, it should be revised. We further require class consistency for practically relevant models because every class should be instantiable in at least one system state or be removed otherwise. Finally, we propose that models should have finitely large states, unless infinite states are an explicit requirement of the modeled domain.

In the remainder of this chapter, we explain how proof obligations can be generated for the different notions of consistency and how HOL-OCL can be used to prove these obligations.

5.2 Consistency Analysis with HOL-OCL

Having formally defined different notions of consistency, we are now able to use these definitions to analyze the consistency of a given UML/OCL model. In this section, we illustrate how consistency analysis can be performed with HOL-OCL. To this end, we show how to generate appropriate proof obligations and how they can be proven in a formal manner.

In contrast to existing analysis approaches that focus on proof automation [Distefano et al., 2000, Gogolla et al., 2005, Jackson et al., 2000], our approach focuses on two features. First, our approach supports exchangeable notions of consistency, i. e., proof obligations for strong consistency can be generated first and if they cannot be proven, they can be replaced by proof obligations for class consistency. In the automatic analysis approaches listed above, one consistency notion is "hard-coded" into the system and in addition, it is typically unclear which notion it is exactly. Second, HOL-OCL offers a sound and complete proof calculus, whereas the above-mentioned automatic approaches are incomplete (cf. Section 2.3.1).

5.2.1 Generating Proof Obligations.

For a given UML/OCL model $M = \{C_1, \ldots, C_n\}$, we can generate proof obligations for strong (5.2), class (5.3), and weak (5.4) consistency. Subtype consistency (5.1) is a necessary prerequisite for the datatype package of HOL-OCL and thus automatically proven during model import [Brucker and Wolff, 2006]. Models that are not subtype-consistent are therefore rejected by HOL-OCL's import mechanism.

Within HOL-OCL, we cannot reason about the meta-level of the model, e. g., proving properties of the set of all classes in the model. Therefore, we instead generate a model-specific instance of the corresponding proof obligations. For strong consistency (5.2), we generate the following proof obligation:

$$\begin{aligned} \exists \tau.\ (\ \tau \vDash C_1 \ \text{::allInstances()} \rightarrow \text{notEmpty()} \wedge \ \ldots \\ \wedge\ \tau \vDash C_n \ \text{::allInstances()} \rightarrow \text{notEmpty()}) \end{aligned} \tag{5.5}$$

The proof obligation (5.6) for class consistency (5.3) requires that for each class in the UML/OCL model, a state τ exist in which the class can be instantiated:

$$\begin{aligned} (\exists \tau_0.\ \tau_0 \vDash C_1 \ \text{::allInstances()} \rightarrow \text{notEmpty()}) \wedge \ \ldots \\ \wedge\ (\exists \tau_{n-1}.\ \tau_{n-1} \vDash C_n \ \text{::allInstances()} \rightarrow \text{notEmpty()}) \end{aligned} \tag{5.6}$$

Finally, the proof obligation (5.7) for weak consistency (5.4) requires that a state τ exist in which at least one class can be instantiated:

$$\begin{aligned} \exists \tau.\ (\ \tau \vDash C_1 \ \text{::allInstances()} \rightarrow \text{notEmpty()} \vee \ \ldots \\ \vee\ \tau \vDash C_n \ \text{::allInstances()} \rightarrow \text{notEmpty()}) \end{aligned} \tag{5.7}$$

The proof obligation for class consistency of the company model thus reads as follows.

```
theorem "∃τ. τ⊨Manager::allInstances()−>notEmpty() ∧
         ∃τ. τ ⊨Employee::allInstances()−>notEmpty() ∧
         ∃τ. τ ⊨Office :: allInstances()−>notEmpty() ∧
         ∃τ. τ ⊨Single :: allInstances()−>notEmpty() ∧
         ∃τ. τ ⊨Cubicle :: allInstances()−>notEmpty()"
```

5.2.2 Proving the Proof Obligations.

In this subsection, we show how consistency proofs can be carried out in HOL-OCL. To this end, we show how to prove the first conjunct of the previously generated proof obligation for the company model, i. e., we prove that the class Manager can be instantiated in some state τ.

The underlying principle of proving consistency with HOL-OCL comprises three steps. First, one has to unfold the definitions related to the class encoding and the user supplied data model. Second, a witness for a consistent state has to be provided. And third, we have to show that this witness fulfills the invariants of the model. Obviously, finding the right witness is the most challenging step in this process.

The proof obligation to express that the class Manager can be instantiated reads as follows.

```
lemma "∃τ. τ ⊨ Manager::allInstances()−>notEmpty()"
```

We start our proof by unfolding the definitions OclAllInstancescompany_Manager_def for the operator ::allInstances() and Manager_def for the class Manager. In HOL-OCL, this is done by applying the simplifier (called simp) with the default set of rules extended by the definitions as explained above.

```
  apply (simp add: OclAllInstancescompany_Manager_def Manager_def)
```

Next, we give a witness, i. e., we instantiate the existential quantifier (*rule_tac* x = ... **in** exI) with an instance of Manager that fulfills its invariants. In our example, such a witness is a manager named "Paul" who does not manage any employees. Paul has a salary of 2000, does not have a manager ($\emptyset$), does not have an office ($\emptyset$), has a budget of 5000, a head count of 2, is a CEO, and does not have any employees ($\emptyset$).

```
  apply (rule_tac x = (λ τ.
         Some(mk_Manager((((OclAny_tag, oid),
              ((Employee_tag, {oid},"Paul", 2000, ∅, ∅),
                (Manager_tag, {oid}, 5000, 2, true, ∅)))))))
 in exI)
```

Finally, we have to show that our witness fulfills the class invariants. In the company model, the class Manager is coarsely constrained by the multiplicities of its associations to Office and Manager, which are inherited from Employee, and to the association to Employee. Furthermore, Manager is constrained by the invariants introduced in the course of this thesis, e. g., *budgetGreaterZero* or *hasManager*. The following command unfolds the invariants and thus evaluates all invariants for Manager against the specified witness.

```
  apply (auto simp: Manager_inv_def)
done
```

After the last simplification, all proof goals are closed and we can finish the proof successfully with the command "**done**."

5.2.3 Using Constraint Patterns in Consistency Proofs.

Since we defined the semantics of our constraint patterns in HOL-OCL, we can use them in consistency proofs for increasing the degree of proof automation. This can be achieved in two steps. First, general statements can be proven for the constraint patterns, e. g., assumptions under which certain pattern instances are consistent or inconsistent. Second, these statements can be added to the simplification tactic of the underlying theorem prover. This allows previously proven knowledge to be automatically applied in future proofs.

We explain these two steps by an example. We formulate a lemma that states that if the multiplicity of some property a is restricted to be less than or equal to some i using an instance of the *Multiplicity Restriction* pattern and i is negative, this implies false.

The lemma requires the assumption that the cardinality of a is greater than or equal to zero, which holds for all objects of type Property because the lower multiplicity bound of each MultiplicityElement must be positive [Object Management Group (OMG), 2006c]. The lemma reads as follows.

```
lemma negativeMultiplicity:
  "[| (τ ⊨(a self)−>size() ≥0;
      (τ ⊨( MultiplicityRestriction  a ≤ i)  self );
      (τ ⊨i  <0)  |]
       ⟹ false"
```

This lemma can be proven by invoking the simplification tactic and unfolding the definition of the *Multiplicity Restriction* pattern.

Having proven lemma negativeMultiplicity, we can use it in subsequent proofs by invoking the following proof command.

```
apply (rule  negativeMultiplicity )
```

Applying the previously proven theorem replaces statements of the form $(\tau \models$ (MultiplicityRestriction a $\geq$ i) self) $\wedge$ $(\tau \models$ i $< 0)$ by false. This saves the proof expert from proving previously proven theorems again. Thus, it increases the degree of automation for future proofs on constraint patterns.

Alternatively, the implication proven in negativeMultiplicity can be added to the simplifier of the theorem prover by invoking the following command. Then, HOL-OCL tries to apply the knowledge from negativeMultiplicity every time the simplification tactic is invoked.

```
declare negativeMultiplicity  [simp]
```

Although constraint patterns can help to increase the level of automation of consistency proofs, proving consistency using provers such as HOL-OCL still requires significant expertise in theorem proving. In the following chapter, we thus present an approach in which previously proven knowledge about constraint patterns is used for an automatic, heuristic consistency analysis.

5.3 Summary

In this chapter, we have investigated consistency concerns for OCL specifications of UML models and have presented distinct formal consistency definitions for OCL. We illustrated how these definitions can be used to generate proof obligations for an interactive theorem prover and showed how these obligations can be formally proven. Furthermore, we have shown how using constraint patterns can contribute to increase the degree of automation in such proofs.

We have seen that consistency analysis by HOL-OCL requires significant expertise in interactive theorem proving. The degree of automation in consistency proofs could be increased by introducing sophisticated proof tactics, which primarily leaves the task of specifying a witness to the proof expert. However, it is questionable whether the degree of automation can be sufficiently high such that specifying a witness is the *only* task that has to be performed manually. In practice, there are two more hurdles for using theorem provers in MDE. Whereas the technical interface between MDE tools and theorem provers represents the smaller hurdle, we have noticed that MDE tools and theorem provers typically have specific platform requirements, e. g., they run on different operating systems. This makes a tight integration of interactive theorem proving in the MDE process difficult.

Thus, an automatic consistency analysis for UML/OCL models is desirable as a means for model developers to get immediate feedback about the consistency of the constraints developed. In the next chapter, we therefore compare existing approaches for consistency analysis of UML/OCL and point out their drawbacks. Subsequently, we introduce a novel approach for analyzing pattern-based constraint specifications that runs fully automatically. This approach ties in with the theorem approach of Section 5.2.3 by using previously proven knowledge about constraints pattern.

Chapter 6

Consistent Model Refinement Using Patterns

UML class models that do not contain textual constraints can be automatically proven to be consistent [Lenzerini and Nobili, 1990, Maraee and Balaban, 2007]. However, if constraint specifications in OCL are added to the models, the consistency of the models and their constraints cannot be decided because OCL is undecidable.

This leaves two choices for consistency analysis: automatic, but incomplete, and interactive, but complete, approaches. In this chapter, we give an overview of different methods for consistency analysis. Subsequently, we introduce a novel, tractable approach for the consistency analysis of pattern-based constraint specifications. This heuristic approach enables incremental consistency analysis, which allows model developers to analyze the model consistency in each refinement step. We develop this approach with an emphasis on *automation* and *tractability*. The approach requires a preceding analysis of the constraint pattern library in which dependencies between the patterns are investigated. For the actual consistency analysis, it suffices to check certain assumptions defined in the dependency analysis, which can be done in polynomial time.

This chapter is structured as follows. In Section 6.1, we give an overview of existing approaches to consistency analysis of UML/OCL models and discuss their advantages and disadvantages. In Section 6.2, we introduce our heuristic approach to consistency analysis based on constraint patterns. In Section 6.3, we analyze dependencies between the *elementary* constraint patterns of our example library from Section 4.3 and thus make the patterns in the library usable for our approach. In Section 6.4, we examine consistency of *composite* constraint patterns. In Section 6.5, we discuss advantages, disadvantages, and limitations of our approach. In Section 6.6, we discuss how our analysis approach integrates into the MDE process. We summarize this chapter in Section 6.7.

6.1 Evaluation of Existing Analysis Approaches

In this section, we first define the criteria that we use for evaluating existing approaches. Subsequently, we evaluate existing analysis approaches and discuss their advantages and disadvantages. We evaluate approaches for the consistency analysis of UML/OCL models by the following criteria.

Coverage Is the modeling language fully covered by this approach, or is only a subset of UML/OCL supported?

Hypothesis What is the hypothesis that is being checked? Typically, analyses use either "model is consistent" or "model is inconsistent" as hypotheses.

Flexibility Is the approach limited to a certain notion of consistency? What effort is necessary for analyzing models for different notions of consistency?

Input What is the necessary input for the analysis? Is there further input required besides the model and the constraint specification?

Output What is the output of the analysis? Typically, this is "yes", "no", or "don't know."

Failure What if the hypothesis cannot be proven? Are there any descriptive messages or counter-examples?

Computational Complexity How expensive is it to compute the result with respect to the size of the model and the constraint specification? Does the analysis always terminate?

Degree of Automation What effort is necessary for the user to use the analysis?

In the following, we provide an overview of existing consistency checking approaches for constraint specifications and subsequently introduce a new approach.

6.1.1 Overview of Existing Analysis Approaches.

In this section, we introduce three existing analysis approaches: *interactive theorem proving*, *witness creation*, and *model checking*.

6.1.1.1 Interactive Theorem Proving.

In this approach, a proof expert uses a theorem prover to carry out consistency proofs. The proof obligations for different notions of consistency can be automatically generated, but the proofs have to be carried out interactively. We know of two implementations of this approach, one for the theorem prover Isabelle/HOL (HOL-OCL, cf. Section 2.6) and one for PVS [Kyas et al., 2005].

In both implementations of this approach, most elements of UML and OCL are covered. Only few elements are not covered, the most important being multiple subtyping in HOL-OCL and partial functions in the PVS-based analysis. The hypothesis used depends on the proof obligation generated. Proof obligations for both "specification is consistent" or "specification is inconsistent" can be automatically generated and subsequently be proven. Thus, this approach offers a high flexibility with respect to different notions of consistency because proof obligations can be generated for any notion of consistency.

The input to this approach is the model, its constraint specification, a generated proof obligation, and a proof. The output of this approach depends on whether the proof obligation could be proven by the user or not. In Isabelle/HOL, successful proofs are finished with the keyword **done**, whereas unsuccessful proof attempts are abandoned by the user with **oops**. In the latter case, a proof obligation for a weaker notion of consistency can be generated, or the hypothesis can be changed. Since this approach is interactive, the effort of using this approach is very high. Proof expertise is required and significant time is necessary to carry out interactive proofs.

6.1.1.2 Witness Creation.

In this approach, the model developer is instructed to specify a witness for the model, i. e., a model state that satisfies all constraints. Typically, the model developer creates several states that are checked against the constraints and the consistency notions. Alternatively, tool support can help the user create the instances.

For example, the user is instructed to create exactly one model instance that contains at least one object per class if strong consistency of the model is required. If this instance does not satisfy the constraints, the user is shown the constraints that are violated and has to modify the model state until it satisfies strong consistency.

There are several tools that validate model states against OCL constraints and support full UML/OCL. These include the tools OSLO [Fraunhofer Fokus, 2007], OCLE [Chiorean et al., 2003], USE [Gogolla et al., 2005], the OCL component of the Eclipse Model Development Tools (MDT) [Eclipse Foundation, 2007a], ITP/OCL [Clavel and Egea, 2006], and the Kent Modeling Framework [Akehurst and Patrascoiu, 2004]. Of all of them, only USE provides a scripting language that can be used to generate a set of model states and automatically search for a witness.

In this approach, all language elements of UML and OCL are supported. Since the core of this approach is finding a witness state, the hypothesis used is that the constraint specification is consistent. The flexibility of this approach is high. Model developers can create different witnesses for different notions of consistency, but they need to be aware of the different notions.

The input to this approach is the model, its constraint specification, and a candidate witness. The output is "yes" if the user-provided model state satisfies the constraint specification. Otherwise, the invariants that are violated by the candidate witness can be displayed and the model developer can adapt the state. Model developers need to invest time and must have domain knowledge, but in comparison to interactive theorem proving, they do not require proof expertise. The weakness of this approach is that the model developer may not be able to find a valid witness although the model is consistent. This can happen if there is a larger witness, i. e., a model state with more objects, than considered by the developer that satisfies the constraints. Such a witness can even contain an infinitely large number of objects (cf. Section 5.1.1.3) and thus, cannot be found with this approach.

6.1.1.3 SAT and Model Checking.

The UML/OCL model and the consistency definitions are transformed into a SAT problem within a predefined size. The problem is then fed to a SAT solver where the problem is either proven or disproven. We know of one such approach using the Alloy Analyzer [Jackson, 2000], which processes models translated using UML2Alloy [Bordbar and Anastasakis, 2005].

A similar approach is to translate UML/OCL into a model checking problem. [Distefano et al., 2000] describes such an approach and provides a mapping of a subset of OCL into a temporal logic called BOTL. For specifications in pure first-order logic, [Reif et al., 2001] shows how a counterexample search can be integrated with theorem proving.

In such approaches, typically all elements of UML, but only a subset of OCL are supported. The hypothesis of the analysis is that the constraint specification is consistent. The flexibility of such approaches is low. Typically, one notion of consistency is hard-coded into the transformation from UML/OCL to the temporal logic.

The input to this approach is the model and its constraint specification. The output is “Yes” if a witness is found or “Don’t know” otherwise. In case no witness can be found, model developers can be shown the model states that violate certain constraints. Subsequently, the model or the constraint specification can be adapted. This approach is fully automated, but the computational complexity is exponential in the size of model elements.

6.1.2 Discussion.

In Figure 6.1, we summarize the different approaches, highlighting the degree of automation and the complexity of each approach. Whereas interactive theorem proving is the only *complete* analysis approach, it offers the least degree of automation when used in the classical way. Note that interactive theorem proving could also be used with semi-decision procedures, which offer a high degree of automation for the cost of completeness. Witness creation also requires user interaction and thus has a similarly low degree of automation. As explained, witness creation approaches are incomplete. The degree of automation is significantly higher in SAT-based approaches and model checking. However, such approaches are incomplete and have exponential complexity. In this figure, we have also positioned the *pattern-based heuristics* approach, which we introduce in the next section.

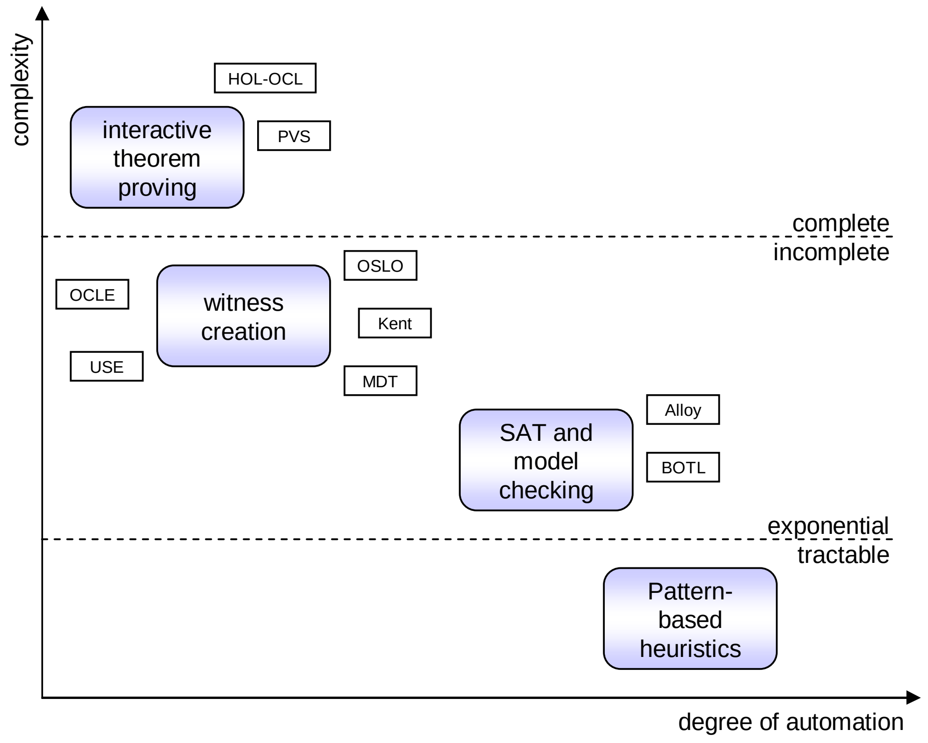


Figure 6.1: Overview of analysis approaches.

In existing approaches to consistency analysis, there is typically no explicit notion of consistency. In contrast, it is often not clear if a model that is shown consistent is strongly-consistent or class-consistent and if it has states with a finite number of objects or not. Furthermore, the consistency definitions are “hard-wired” into the tools, i. e., tools that check strong consistency cannot easily be changed to check class consistency. An exception to this rule is HOL-OCL because the desired consistency notion can be explicitly stated as proof obligation.

In practice, automatic approaches based on model checking are desirable during the development phase because they allow developers to easily check their models for consistency without additional effort by the developer. However, they have one major disadvantage: Certain types of inconsistencies cannot be detected. For instance, if a model has states with an infinitely large number of objects only, a model checking approach will only generate states with a finite number of objects and then report "Don't know." In addition, such approaches have an exponential complexity and can cause significant interruptions of the user's workflow.

In the subsequent section, we introduce a novel heuristic analysis approach based on constraint patterns that automatically analyzes constrained class models. We intend to develop this approach with three advantages over the other analysis approaches. First, this approach runs fully automatically when invoked by the model developer and thus does not need user interaction such as interactive theorem proving and witness creation. Second, constraint specifications can be *efficiently* checked in a Computer Aided Software Engineering (CASE) tool whether its constraints are in some consistent or inconsistent subset (otherwise, we "don't know") unlike SAT-based approaches, which have exponential complexity. Third, this approach allows for detecting certain cases in which a model has infinite states only, which can usually not be achieved in witness creation and SAT-based approaches.

6.2 Heuristic Analysis Based on Pattern Theorems

In this section, we introduce a novel approach to consistency analysis of pattern-based constraint specifications. In general, this approach comprises proving general consistency properties of the constraint patterns *once* and subsequently use this knowledge for an automatic and efficient heuristic analysis on pattern instances within the development tool.

6.2.1 Consistency Theorems.

We capture the general consistency properties of a pattern Π as a set of assumptions under which adding an instance of Π to the constraint specification preserves the respective consistency of the specification. This analysis has to be done only once for each constraint pattern in a given pattern library. As a result, we formulate and prove a *consistency theorem* for each pattern.

Definition 20 (Consistency Theorem). *A* consistency theorem *for a constraint pattern* Π *comprises sufficient syntactic conditions* $P_1, \ldots, P_n$ *under which adding an instance of* Π *to a consistent model* M *and its constraint specification* Φ_M *preserves the* { *weak, class, strong* } *consistency of the model. It has the following structure:*

- *Assume* $\langle M, \Phi_M \rangle$ *x-consistent. Let* ψ *be an instance of pattern* Π.
- *Assume properties* $P_1(M, \Phi_M), \ldots, P_n(M, \Phi_M)$ *about the model and its constraint specification.*
- *Then,* $\langle M, \Phi_M \cup \{\psi\} \rangle$ *x-consistent.*

When the patterns are used, it is sufficient to check whether the assumptions $P_1, \ldots, P_n$ hold. Since these assumptions are syntactic properties of the UML/OCL model $\langle M, \Phi_M \rangle$, they can be checked in polynomial time. In our approach, we assume that the initial class model without OCL constraints is consistent, which is a decidable property that can be computed in linear time [Maraee and Balaban, 2007].

The main challenge in this approach is identifying which patterns can potentially contradict a given pattern Π. To this end, we divide elementary constraint patterns into two types of patterns, namely those that restrict the value of attributes, e. g., *Unique Identifier*, and those that restrict the structure of the object graph spanned by objects and links between them, e. g., *Surjective Association*. It is a basic observation that constraint patterns of one kind are independent of constraint patterns of the other kind, i. e., an instance of a pattern constraining an attribute value can never contradict an instance of a pattern constraining the object graph. The only exception is the *Multiplicity Restriction* pattern, which relates the multiplicity of an association to an attribute value and thus restricts both an attribute value and the object graph. This distinction helps us to determine which patterns can potentially contradict a given pattern Π.

We use the *No Cyclic Dependency* pattern to illustrate consistent model refinement using constraint patterns. To this end, we state the following theorem that defines the necessary assumptions under which a constrained model remains strongly-consistent after instantiating this pattern. We chose to require strong consistency in the consistency theorems in this thesis because the proofs for this notion of consistency are more compact and illustrative. The theorems can be easily adapted to other notions of consistency though, e. g., class consistency. After the proof for the following theorem, we provide a variant of the theorem for class consistency.

Theorem 1. *Let M be a model and Φ_M be a strongly-consistent constraint specification based on patterns, ϕ be an instance of the* No Cyclic Dependency *pattern with class C as context and navigation path $p_1.p_2. \ldots .p_n$. Further assume there is no instance of the* Object In Collection *pattern in Φ_M on any element of the path $p_1.p_2. \ldots .p_n$. If there are $j, k \in \{1, \ldots, n\}$ such that*

(i) the lower multiplicity bound of p_j and p_k^{-1} in M is zero and

(ii) there is no instance of the Surjective Association *or* Bijective Association *pattern in Φ_M with parameter value p_k, and*

(iii) there is no instance of the Multiplicity Restriction *pattern in Φ_M with p_j and p_k^{-1} as values for any parameter,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

In the following, we sketch the ideas behind the assumptions of this theorem and show a full proof of this theorem in Section 6.3. Because of the requirement that the initial model is strongly-consistent, there exists a state τ in which each class of M is instantiated. If τ does not satisfy the new constraint ψ, we construct a state τ' from τ as follows. First, we delete the j^{th} link from the cycle. Now, $\tau' \models \phi$, but τ' may not satisfy the multiplicity constraints in Φ_M. Thus, we walk the path backwards starting from position j and instantiate and link new objects until the multiplicity constraints hold. This will eventually be the case because of the existence of p_k and its properties that we assume. Thus, $\tau' \models \Phi_M$ and M and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

The assumptions (i)-(iii) are crucial for the correctness of above construction. Assumption (i) requires the existence of an association end p_j on the constrained path that has a lower multiplicity bound of zero, and there must be an association end p_k whose inverse association end has a lower multiplicity bound of zero. Generally speaking, this assumption allows us to delete existing links in the above construction and ensures its termination.

However, it is not sufficient that the lower multiplicity bound of these association ends is zero. In addition, there must be no constraints in the constraint specification Φ_M of the model M that constrain the lower multiplicity bound of these association ends. There are three constraint patterns in our library that can be used to restrict lower multiplicity bounds: *Surjective Association*, *Bijective Association*, and *Multiplicity Restriction*. Therefore, we added assumptions (ii) and (iii) to the consistency theorem. In addition, we exclude that the reflexive path is constrained by an *Object In Collection* constraint, which would otherwise affect these association ends.

The theorem for the *No Cyclic Dependency* pattern for class consistency is almost identical to the theorem for strong consistency with the exception that the term "strongly-consistent" is replaced by "class-consistent". The theorem reads as follows.

Theorem 2. *Let M be a model and Φ_M be a class-consistent constraint specification based on patterns, ϕ be an instance of the* No Cyclic Dependency *pattern with class C as context and navigation path $p_1.p_2. \dots .p_n$. Further assume there is no instance of the* Object In Collection *pattern in Φ_M on any element of the path $p_1.p_2. \dots .p_n$. If there are $j, k \in \{1, \dots, n\}$ such that*

(i) the lower multiplicity bound of p_j and p_k^{-1} in M is zero and

(ii) there is no instance of the Surjective Association *or* Bijective Association *pattern in Φ_M with parameter value p_k, and*

(iii) there is no instance of the Multiplicity Restriction *pattern in Φ_M with p_j and p_k^{-1} as values for any parameter,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is class-consistent.

Whereas the theorems for strong consistency and class consistency are almost identical, the proofs require some adjustments. Since our basic assumption is now the *class consistency* of the constraint specification, there exists a *set of states* $T = \{\tau_1, \dots, \tau_n\}$ in which each class of M is instantiated. For each $\tau_i \in T$ that does not satisfy ϕ, we perform the construction that we introduced in the proof for strong consistency for τ_i. As a result, we obtain a set of states T' in which each τ satisfies both Φ_M and ϕ. Thus, it is a witness for the class consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$.

We are interested in *sufficient* conditions that pattern instances preserve the consistency of the model. Thus, the theorems typically have the form $\mathcal{P} \Rightarrow Q$, where $\mathcal{P}$ is a set of syntactic assumptions and Q is the fact that the refined model is consistent. In general, it is not possible to have consistency theorems in the form $\mathcal{P} \Leftrightarrow Q$ because the semantics of the patterns is based on OCL, which is an undecidable language.

6.2.2 Using Consistency Theorems for Analysis.

For analyzing pattern-based constraint specifications for consistency, it is sufficient to check for each pattern instance whether the assumptions defined in the respective pattern theorem hold, which consists of syntactic checks that can be performed in polynomial time. We will see in the next section that the complexity for checking the assumptions for one consistency theorem is $\mathcal{O}(|\Phi| \cdot |P|)$, where $|\Phi|$ is the number of pattern instances in the constraint specification and $|P|$ is the length of the longest path used as parameter value for the pattern instances.

For each pattern instance in the constraint specification, consistency analysis can either result in *consistent* if the assumptions in the respective theorem hold or *don't know* otherwise. In the latter case, the consistency of the constraint specification must be proven by different means as described in Section 6.1.1. Note that using consistency theorems as defined in Definition 20, the analysis cannot result in *inconsistent*. However, our approach can be extended by a complementary set of theorems that state the assumption under which a given pattern instance *violates* the consistency of a model. We consider this future work and thus illustrate it further in Section 9.2.

Besides using consistency theorems for an automatic analysis, they can also be used for increasing the degree of automation in interactive consistency proofs. As explained in Section 5.2.3, theorems can be used as simplification rules once they are proven. This can simplify future proofs of theorems that contain statements that can be inferred from previously proven theorems or that contain previously proven theorems as proof goals.

In the next section, we state and prove consistency theorems for each constraint pattern from Section 4.3 in detail. Note that we do not carry out the proofs in HOL-OCL using the HOL-OCL semantics of the constraint patterns because such formal proofs would require a deep embedding (cf. [Boulton et al., 1992]) of the pattern semantics and the consistency notions in contrast to the current shallow embedding (cf. Section 2.6.2). Such a deep embedding, in turn, is beyond the scope of this thesis.

6.3 Dependencies between Elementary Patterns

In this section, we investigate under which assumption instances of the constraint patterns presented in Section 4.3 preserve the consistency of a model. Starting from a model M and a strongly-consistent constraint specification Φ_M, we add instances of our constraint patterns and analyze potential conflicts. We assume that all pattern instances are not negated and that there are no instances of the *Literal OCL* pattern, which contains arbitrary OCL expressions. We discuss these limitations in Section 6.5 where we also elaborate on using the theorems about elementary patterns for analyzing composite patterns and scaling this approach when new patterns are added.

We use the example model in Figure 6.2, which represents a simple class model with classes, attributes, associations, and generalization, for illustrating our findings — our consistency theorems make statements about *arbitrary* class models of course.

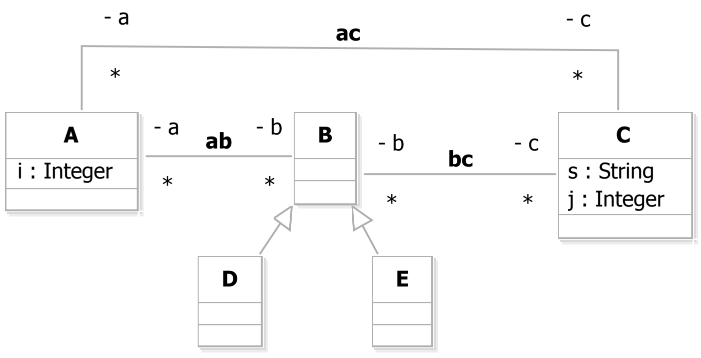


Figure 6.2: Generic class diagram.

6.3.1 Association-Restricting Patterns.

6.3.1.1 *No Cyclic Dependency*.

The *No Cyclic Dependency* pattern can be instantiated to disallow cyclic links between objects on a certain navigation path. We defined it as follows in Section 4.3.

```
pattern NoCyclicDependency(property: Sequence(Property)) =
 self.closure(property)−>excludes(self)

pattern closure(property: Sequence(Property)) =
  self.property−>union(self.property.closure(property))
```

If the pattern is instantiated on a navigation path p, it has to be ensured that the multiplicities of the association ends included in p allow noncyclic instantiations, as expressed by the following theorem.

Theorem 3. *Let $\langle M, \Phi_M\rangle$ be a strongly-consistent model, ϕ be an instance of the* No Cyclic Dependency *pattern with class C as context and navigation path $p_1.p_2.\ \ldots\ .p_n$. Further assume there is no instance of the* Object In Collection *pattern in Φ_M on any element of the path $p_1.p_2.\ \ldots\ .p_n$. If there are $j, k \in \{1, \ldots n\}$ such that*

(i) the lower multiplicity bound of p_j and p_k^{-1} is zero and
(ii) there is no instance of the Surjective Association *pattern or* Bijective Association *pattern in Φ_M with parameter value p_k, and*
(iii) there is no instance of the Multiplicity Restriction *pattern in Φ_M with p_j and p_k^{-1} as values for any parameter,*

then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M\rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. We distinguish two cases:

Case 1: $\tau \models \Phi_M \wedge \phi$. In this case, τ does not contain a cyclic link between objects of class C on path $p_1.p_2.\ \ldots\ .p_n$. Thus, $\tau' = \tau$.

Case 2: $\tau \not\models \Phi_M \wedge \phi$. In this case, there is an object $o_1 : C \in \tau$ and a sequence $(o_1, \ldots, o_m, o_1)$ of objects that represent a cyclic link in which the link from object o_{i-1} to o_i is an instance of association end p_i. We construct τ' from τ by deleting the link from o_{j-1} to o_j. This deletion does not violate any invariant of C_{j-1} because our assumptions state that no relation between objects of class C_{j-1} and C_j is required by a *Object In Collection* constraint, and objects of class C_{j-1} are not required to relate to objects of class C_j because the lower multiplicity bound of association end p_j is zero (i) and not further constrained (iii). Now, $\tau' \models \phi$.

If after the deletion of the link $\tau' \models \Phi_M$ holds, the construction is finished. If not, the deletion has violated the multiplicity constraints of at least one class of which an object participated in the cycle. In this case, we initially create an object o'_{j-1} of type C_{j-1} and link it to o_j. Counting an index i from $j-1$ down to 1 (and potentially from m to $j+1$ afterwards), we create an object of type C_{i-1} and link it to o_i only if class C_i requires a link to class C_{i-1}; otherwise, the algorithm terminates. It eventually terminates because there exists a class C_k that does not require a link to class C_{k-1} because the lower multiplicity bound of the opposite association end of p_k can be zero and is not further constrained (i-iii), and there is no *Object In Collection* constraint on p_k. After this construction, the multiplicity constraints hold that were violated by the deletion and thus, $\tau' \models \Phi_M$, and, as shown before, $\tau' \models \phi$. Thus, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent. □

Figure 6.3 shows the construction that we use in the proof by an example. In this example, there exists a state τ in which there is a cyclic link between the objects. We construct a state τ' according to the construction in the proof; the association end b is the required p_j and c the required p_k.

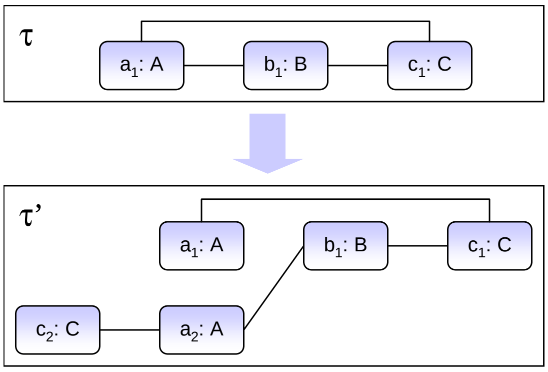


Figure 6.3: Illustration of the proof for Theorem 3.

An example of *inconsistent* pattern instances reads as follows.

```
context A
   inv: NoCyclicDependency(b.c.a)
   inv:  MultiplicityRestriction (b,>=,1)
context B
   inv:  MultiplicityRestriction (c,>=,1)
context C
   inv:  MultiplicityRestriction (a,>=,1)
```

These constraints are inconsistent because the first invariant disallows cycles on the path b.c.a whereas the remaining invariants require each object on the path to be connected to at least one successive object. Thus, model states can either contain zero or infinitely many objects of classes A, B, and C, which violates strong consistency.

6.3.1.2 *Object In Collection.*

The *Object In Collection* pattern can be instantiated to require objects of a class to be contained in a set of related elements. We defined it as follows.

```
pattern ObjectInCollection(set:Sequence(Property), element:Sequence(Property)) =
  self.set->includes(self.element)
```

In Theorem 3, we stated dependencies between this pattern and the *No Cyclic Dependency* pattern. In addition, there are further dependencies, as stated in the following theorem.

Theorem 4. *Let $\langle M, \Phi_M\rangle$ be a strongly-consistent model, ϕ be an instance of the* Object In Collection *pattern with context class C, set $= p_1.p_2. \ \ldots \ .p_m$, and element $= p_1.p_2. \ \ldots \ .p_n$. If*

(i) there is no instance of the No Cyclic Dependency *pattern in Φ_M with parameter property = set and*

(ii) the upper multiplicity bound of p_m is at least one, and

(iii) there is no instance of the Multiplicity Restriction *pattern in* Φ_M *with any* $p_i \in \{p_1, \dots, p_m\}$ *as value for the navigation parameter,*

then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ *is strongly-consistent.*

Proof. Because $\langle M, \Phi_M\rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models \phi$.

For each object o_0 of type C for which the *Object In Collection* constraint does not hold, we perform the following construction, which we illustrate in Figure 6.4. Starting from o_0, we navigate along the path $p_1.p_2. \ \dots \ .p_m$. This path ends at an object o_k because of $\tau \not\models \phi$. For all $i, k < i < m$, create an object o_i of class $type(p_i)$ and link it to the previous object. This does not violate any constraints because at least one relation between these objects can exist because their multiplicities are not constrained (iii). Connect the last object of $type(p_{m-1})$ to o_0. This does not violate any constraints because arbitrary many objects of type C can be connected to objects of type $type(p_{m-1})$ because the multiplicity of p_m is at least one (ii). The last step creates a cyclic link between o_0 and itself, which does not violate any constraint because cycles are not forbidden (i). Due to this cycle, $\tau' \models \phi$ and thus, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent. □

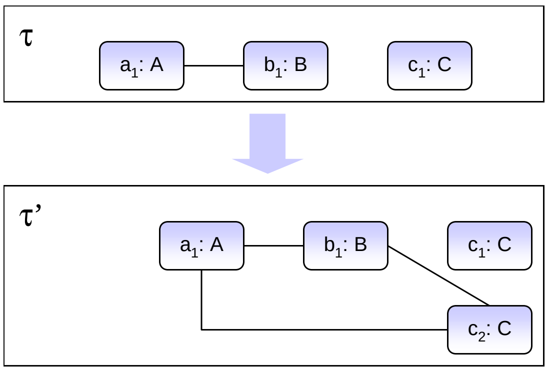


Figure 6.4: Illustration of the proof for Theorem 4.

An example of *inconsistent* pattern instances reads as follows.

```
context A
inv: ObjectInCollection(b.c.a)
inv: NoCyclicDependency(b.c.a)
```

These invariants are inconsistent because the first invariant requires each object of class A to be in the set of objects reachable via path b.c.a, which implies a cycle. However, the second invariant explicitly forbids such cycles.

6.3.1.3 *Surjective Association.*

The *Surjective Association* pattern can be instantiated to make an association end surjective. We defined it as follows.

```
pattern SurjectiveAssociation(property:Sequence(Property)) =
  self.property.allInstances()−>forAll ( y |
     self.allInstances()−>exists( x | x.property−>includes(y)))
```

We define the consistency theorem for the *Surjective Association* pattern as follows.

Theorem 5. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Surjective Association *pattern with class C as context and navigation path $P = p_1.p_2. \ldots .p_n$. If*
(i) for each $1 \leq i \leq n$, the upper multiplicity bound of property p_i^{-1} is greater than zero,
(ii) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts any opposite association end p_i^{-1} for all $i, 1 \leq i \leq n$, and*
(iii) there is no instance of the No Cyclic Dependency *pattern in Φ_M with navigation path $p_1.p_2. \ldots .p_n$*
then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models \phi$.

In this case, there is an object y of class $type(p_n)$ in τ that is not linked to an object of class C via path P. We establish a link between an object of class C and y as follows. From y, we navigate for each $1 \leq i < n$ backwards on path P. For each part i of the path for which there does not exist a link, we either create an instance of p_i between object o_{i+1} and an existing object o_i if the multiplicity constraints of class $type(o_i)$ allow o_i be connected to o_i or otherwise, we create a new object o'_i of class $type(o_i)$. Creating such a link is possible because objects of class $type(o_{i+1})$ can be connected to objects of $type(o_i)$ since the multiplicity of association end p_i^{-1} is at least one (i) and not further restricted by an *Multiplicity Restriction* (ii). Since we try to connect to an existing object, a cycle can be introduced, but because cycles on this part are not forbidden (iii), this construction does not violate Φ_M. This construction terminates after the n^{th} step connecting an object of type C to a path that leads to y. Therefore, $\tau' \models \phi$, and since this construction has not violated any constraint in Φ_M, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

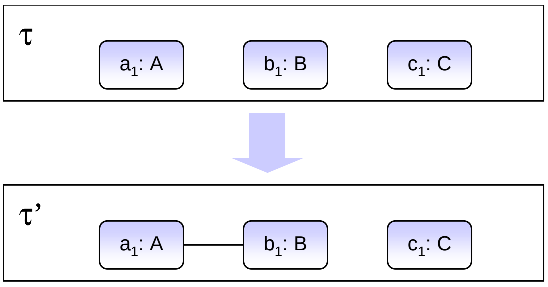


Figure 6.5: Illustration of the proof for Theorem 5.

An example of *inconsistent* pattern instances reads as follows.

```
context A
   inv: SurjectiveAssociation(b)
context B
   inv:  MultiplicityRestriction  (a,<,1)
```

Whereas the first invariant requires every object of class B to be connected to an object of class A, the second invariant forbids this by stating that no objects of class A may be connected to objects of class B.

6.3.1.4 *Injective Association.*

The *Injective Association* pattern can be instantiated to make an association end injective. We defined it as follows.

```
pattern InjectiveAssociation(property:Sequence(Property)) =
  self.property->size() = 1 and
  self.allInstances()->forAll (x,y | x.property = y.property implies x=y)
```

We define the consistency theorem for the *Injective Association* pattern as follows.

Theorem 6. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Injective Association *pattern with class C as context and navigation path $P = p_1.p_2. \ldots .p_n$. If*

*(i) the upper upper multiplicity bound of the opposite association end of each $p_i \in \{p_1, \ldots, p_n\}$ is either one or * and*

(ii) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts the opposite association end of any $p_i \in \{p_1, \ldots, p_n\}$,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models \phi$.

In this case, there are two or more objects of class C in τ that are connected to the same object of class $type(p_n)$ as illustrated in Figure 6.6. We construct τ' from τ as follows. For every object o_n of class $type(p_n)$, we walk the path P backwards. For each $i, 1 \leq i < n$, if there is more than one link to from o_i to o_{i+1}, we nondeterministically delete all but one link. This preserves the invariants of the classes on the "right-hand side" of the link, because exactly one link will be left and the multiplicity of this association is either one or unlimited (*) (i) and not further constrained (ii). After this construction, there is at most one link from an object of class C to an object of class $type(p_n)$ and thus, $\tau' \models \phi$.

However, the multiplicities of the classes on the "left-hand side" of the deleted links may have been violated by the previous construction. We repair the multiplicities as follows. For each object of class C, we walk path P from $1 \leq i < n$. If the multiplicity invariants of class $type(p_i)$ are violated, we create objects of class $type(p_{i+1})$ until the invariants are satisfied. Since each newly created object o_i is connected to exactly one object o_{i-1}, $\tau' \models \phi$ still holds. After this second part of the construction, the all multiplicity constraints hold and thus, $\tau' \models \Phi_M$. Because $\tau' \models \phi$ also holds, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

An example of *inconsistent* pattern instances reads as follows.

```
context A
   inv: InjectiveAssociation(b)
context B
   inv:  MultiplicityRestriction (a,>,1)
```

Whereas the first invariant states that no object of class B may be connected to more than one object of class A, the second invariant states the exact opposite.

6.3.1.5 *Bijective Association.*

The *Bijective Association* pattern can be instantiated to make an association bijective. We defined it as follows.

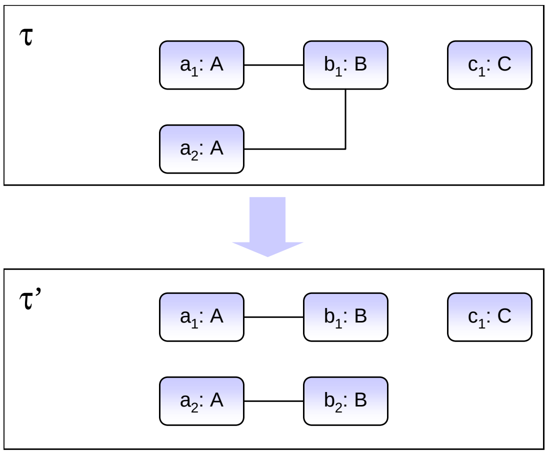


Figure 6.6: Illustration of the proof for Theorem 6.

```
pattern BijectiveAssociation (property:Sequence(Property)) =
  InjectiveAssociation (property) and
  SurjectiveAssociation(property)
```

We define the consistency theorem for the *Bijective Association* pattern as follows.

Theorem 7. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Bijective Association *pattern with class C as context and navigation path $P = p_1.p_2. \ \ldots \ .p_n$. If*
(i) for each $1 \leq i \leq n$, the upper multiplicity bound of property p_i^{-1} is greater than zero,
(ii) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts any the opposite association end p_i^{-1} for all $1 \leq i \leq n$,*
(iii) there is no instance of the No Cyclic Dependency *pattern in Φ_M with navigation path $p_1.p_2. \ \ldots \ .p_n$,*
*(iv) the upper upper multiplicity bound of the opposite association end of each $p_i \in \{p_1, \ldots, p_n\}$ is either one or * and*
(v) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts the opposite association end of any $p_i \in \{p_1, \ldots, p_n\}$,*
then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Note that assumptions (i)-(iii) are the assumptions from the *Surjective Association* pattern and assumptions (iv) and (v) are the assumptions from the *Injective Association* pattern.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models \phi$.

We construct a state τ' from state τ by making the navigation path $P = p_1.p_2. \ \ldots \ .p_n$ both surjective and injective. Making this navigation path surjective is possible because of the assumptions (i)-(iii) (cf. Theorem 5) and (iv)-(v) (cf. Theorem 6). After this construction, P is both surjective and injective. Thus, P is bijective and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

An example of *inconsistent* pattern instances reads as follows.

```
context A
   inv: BijectiveAssociation(b)
context B
   inv:  MultiplicityRestriction  (a,>,1)
```

These invariants are inconsistent because the first invariant entails a one-to-one relation between objects of classes A and B whereas the second invariant imposes a one-to-many relation between objects of B and objects of A.

6.3.1.6 *Type Restriction.*

The *Type Restriction* pattern can be used to restrict an association that is defined between some class and some superclass by limiting the allowed subclasses. We defined it as follows.

```
pattern TypeRestriction(property:Property, allowedClasses:Set(Class)) =
  self.property->forAll(x | allowedClasses->exists(t | x.oclIsTypeOf(t)))
```

We define the consistency theorem for the *Type Restriction* pattern as follows.

Theorem 8. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Type Restriction *pattern with class C as context, a navigation path $P = p_1.p_2. \ \ldots \ .p_n$ and a set $S = \{C_1, \ldots, C_n\}$ of allowed classes. If*

(i) for each $1 \leq i \leq n$, the lower multiplicity bound of property p_i^{-1} is zero,

(ii) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts any the opposite association end p_i^{-1} for all $1 \leq i \leq n$,*

(iii) there is no instance $\psi(p', S')$ of the Type Relation *pattern in Φ_M where p' is a suffix of p and $S' - S \neq \emptyset$, and*

(iv) there is no other instance $\psi(p', S')$ of the Type Restriction *pattern in Φ_M where p' is a suffix of p and $S' \cap S \neq \emptyset$,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exists a path between an object o_1 of class C to an object o_n with $class(o_n) \notin \{C_1, \ldots, C_n\}$. We construct τ' by deleting all links between objects of class $type(p_n^{-1})$ and o_n as shown in Figure 6.7. This does not violate the multiplicity constraints of $class(o_n)$ because this class can be related to zero objects of class $type(p_n^{-1})$ because the lower multiplicity bound of p_n^{-1} is zero (i) and not further constrained (ii). The deletion of the link does not violate any *Type Relation* constraint because no object of $class(o_n)$ is required to be on path P (iii). Now, $\tau' \models \phi$.

If the multiplicity constraints of class $type(p_n^{-1})$ are violated, we create objects of any allowed class $C_i \in \{C_1, \ldots, C_n\}$ until the multiplicity constraints of class $type(p_n^{-1})$ are satisfied. Furthermore, the newly created objects do not violate any *Type Restriction* constraint in Φ_M because there is no other type restriction that requires any class not in $\{C_1, \ldots, C_n\}$ (iv). Now, also $\tau' \models \Phi_M$ holds and thus, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

The following invariants applied to the model in Figure 6.2 are not strongly-consistent because no instance of D can be created.

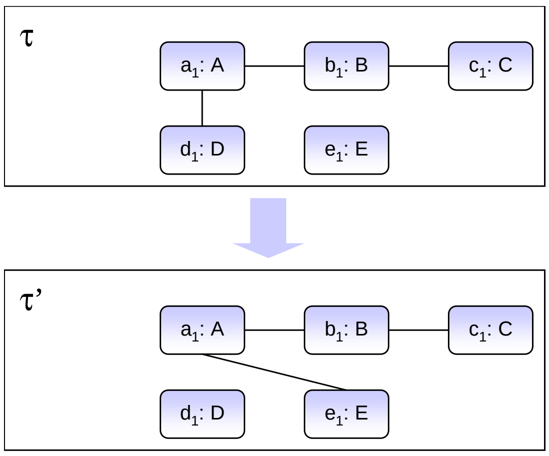


Figure 6.7: Illustration of the proof for Theorem 8.

```
context A
   inv: TypeRestriction(b,E)
context D
   inv:  MultiplicityRestriction (a,>,0)
```

In particular, the first invariant requires that only objects of the subclass E of B may be connected to objects of class A on the association end b. However, the second invariant requires that objects of class D, the other subclass of B, must be connected to at least one object of class A, which contradicts the first invariant.

6.3.1.7 *Type Relation.*

The *Type Relation* pattern can be used to enforce that instances of certain subclasses $C_1, \dots, C_n$ of C_0, the requiredClasses, must participate in some relation. We defined the pattern as follows.

```
pattern TypeRelation(property:Sequence(Property), requiredClasses:Set(Class)) =
  requiredClasses−>forAll(c | self.property−>exists(p | p.oclIsTypeOf(p)))
```

We define the consistency theorem for the *Type Relation* pattern as follows.

Theorem 9. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Type Relation *pattern with class C as context, a navigation path $P = p_1.p_2. \ \dots \ .p_n$ and a set $S = \{C_1, \dots, C_n\}$ of required classes. If*

(i) there exists a $p_i \in \{p_1, \dots, p_n\}$ for which the upper multiplicity is greater than or equal to $|S|$,

(ii) there is no instance of the Multiplicity Restriction *pattern in Φ_M that restricts the above-mentioned association end p_i, and*

(iii) there is no instance $\psi(p', S')$ of the Type Restriction *pattern in Φ_M where p' is a suffix of p and $S' - S \neq \emptyset$,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there is an object $o_1 : C$ that is not linked to an object of class $C_j \in S$. Walk the path P from o_1 to p_i. At p_i, create a new object o_i' of class $type(p_i)$ and link it to the previous object in the path as illustrated in Figure 6.8. After this construction, $\tau' \models \Phi_M$ still holds because the unlimited multiplicity of p_i (i), which is not further constrained (ii),allows one to connect an unlimited number of elements to the previous objects in the path.

From o_i', continue to walk path P, creating a new object in each step. The last object in the path must be of type C_j. This is possible because objects of this class are not forbidden to connect to this path by any instance of the *Type Restriction* pattern (iii). Then, $\tau' \models \phi$, and thus, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent. □

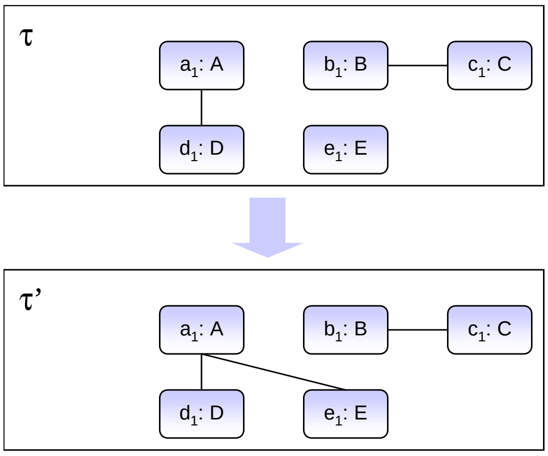


Figure 6.8: Illustration of the proof for Theorem 9.

The following set of invariants is inconsistent.

```
context A
   inv: TypeRelation(b,{D,E})
   inv:  MultiplicityRestriction (b,<=,1)
```

The first invariant requires every object of class A to be related to objects of class D *and* objects of class E, whereas the second invariant allows objects of class A to be related to at most one object of class B, the superclass of D and E.

6.3.1.8 *Unique Path.*

The *Unique Path* pattern can be used to limit the number of links between two objects to one. We defined it as follows.

```
pattern UniquePath(property: Sequence(Property)) =
  self.property->exists(m1,m2 | m1->closure(property)->intersect(
                                m2->closure(property))->notEmpty()
                          implies m1=m2)
```

We define the consistency theorem for the *Unique Path* pattern as follows.

Theorem 10. *Let $\langle M, \Phi_M\rangle$ be a strongly-consistent model, ϕ be an instance of the* Unique Path *pattern with class C as context and a navigation path $P = p_1.p_2. \ \ldots \ .p_n$. If there are no $p_i, p_j \in P$ such that*

(i) the upper multiplicity bound of p_i and p_j^{-1} is greater than one and

(ii) there is an instance of the Multiplicity Restriction *pattern in Φ_M on p_i or p_j^{-1},*
(iii) there is no instance of the Injective Association *or* Bijective Association *pattern on P or P^{-1} in Φ_M,*

then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M\rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there is more than one path between an object o_0 of class C and an object o_n of class $type(p_n)$. We start walking past the objects o_i on path P starting from object o_0. If object o_i has more than one link to objects of class C_{i+1}, we nondeterministically delete one link, as illustrated in Figure 6.9. This does not violate the multiplicity constraints of class C_i of object o_i because o_i is still connected to at least one object of class C_{i+1} and the upper multiplicity bound of p_i is at most one (i, ii). Now, $\tau' \models \phi$.

However, the multiplicity constraints of the object o_{i+1} at the opposite end of the deleted link may be violated. We thus create a new object o_i' of class C_i and connect it to o_{i+1}. Now, the multiplicity constraints for o_{i+1} are satisfied. If the multiplicity constraints of o_i' are not satisfied, we create the objects with which o_i' is required to have relations and connect them to o_i'. Object o_n is now connected to two different objects of class C, which is legal because path P is not required to be injective (iii). Now, also $\tau' \models \Phi_M$ holds and thus, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent. □

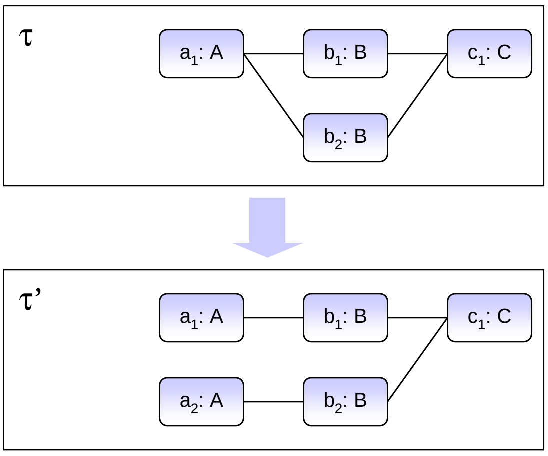


Figure 6.9: Illustration of the proof for Theorem 10.

The following set of invariants is inconsistent.

```
context A
  inv: UniquePath(b.c)
  inv:  MultiplicityRestriction (b,=,2)

context B
  inv:  MultiplicityRestriction (a,=,1)
  inv:  MultiplicityRestriction (c,=,1)

context C
  inv:  MultiplicityRestriction (b,=,2)
  inv: InjectiveAssociation (b.a)
```

The first invariants limits the number of allowed paths between objects of class A and class C to one. However, all *Multiplicity Restriction* constraints enforce a diamond configuration in combination with the *Injective Association* constraint, which contradicts the first invariant.

6.3.1.9 *Path Depth Restriction.*

The *Path Depth Restriction* pattern can be used to limit instances of reflexive associations to a certain length. We defined it as follows.

```
pattern PathDepthRestriction(property: Sequence(Property), maxDepth:Integer) =
  self.pathDepthSatisfied(property,maxDepth−1,0)

pattern pathDepthSatisfied(property: Sequence(Property), max:Integer, counter:Integer) =
   if  (counter > max or counter < 0 or max < 0) then false
  else  if  (self.property−>isEmpty()) then true
        else  self.property−>forAll(m|m.pathDepthSatisfied(property, max, counter+1))
        endif
  endif
```

We define the consistency theorem for the *Path Depth Restriction* pattern as follows.

Theorem 11. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Unique Path *pattern with class C as context, a navigation path $P = p_1.p_2. \ldots .p_n$, and a maximum depth of n. $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.*

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there is an instance of path P in τ of length m with $m > n$. We know that the path has finite length because $\langle M, \Phi_M \rangle$ is strongly-consistent. Thus, we know that instances of this path with finite length can exist.

We create a state τ' from τ as follows. Starting from the first element o_0 on the path of class C, we follow the path instance n times, ending up at another object o_i of class C. We cut the path by deleting the link between o_i and o_{i+1}, as shown in Figure 6.10. Next, we restore the head of the remainder of the path by instantiating a new object of class C and linking it to o_{i+1}. We recursively apply these steps until the end of the path is reached. With this construction, we have split the instance of the path into parts with a maximum size of n each and thus, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

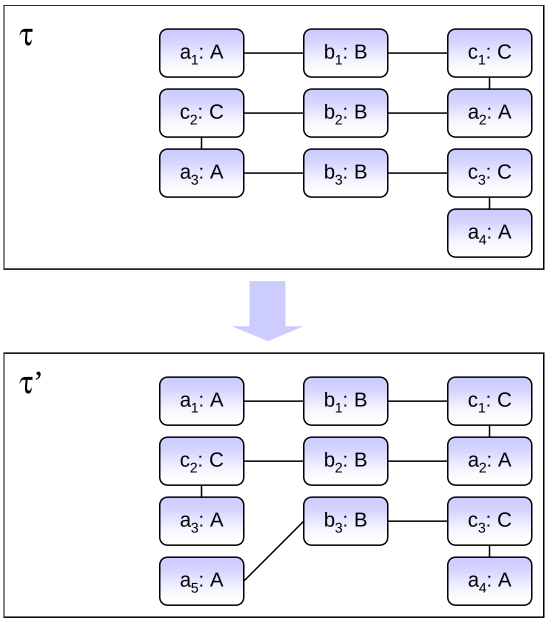


Figure 6.10: Illustration of the proof for Theorem 11.

6.3.2 Attribute-Restricting Patterns.

Values of two or more attributes can be mutually dependent. Consider the following example.

```
context A
  inv: self.b−>forAll( b | self.x > b.y )  −− Attribute Relation

context B
  inv: self.y = self.c−>sum(z)  −− AttributeSumRestriction

context C
  inv: self.z = self.a.x  −− AttributeValueRestriction
```

In every model state, each summand of the sum restriction for class B is greater than the sum. This is a contradiction and thus, no satisfying instance exists. We reflect this in the following consistency theorems by not warranting consistency if attributes are restricted by more than one constraint.

6.3.2.1 *Attribute Relation.*

Using the *Attribute Relation* pattern, attributes can be related to other attributes. We defined this pattern as follows.

```
pattern AttributeRelation (navigation:Sequence(Property), remoteAttribute:Property,
                                        operator: OclExpression, contextAttribute:Property) =
self.navigation−>forAll( x | x.remoteAttribute operator contextAttribute)
```

We define the consistency theorem for the *Attribute Relation* pattern as follows.

Theorem 12. *Let* $\langle M, \Phi_M \rangle$ *be a strongly-consistent model,* $\phi(navigation, remote\text{-}Attribute, op, contextAttribute)$ *be an instance of the* Attribute Relation *pattern with class* C *as context. If* *(i) there is no instance of the* Attribute Sum Restriction, Attribute Value Restriction, Multiplicity Restriction, *or another* Attribute Relation *pattern in* Φ_M *in which* $contextAttribute$ *is used as a parameter,*

(ii) $remoteAttribute \neq contextAttribute$*, and*

(iii) there is no instance of the Unique Identifier *pattern in* Φ_M *in which* $contextAttribute$ *is one of the unique properties,*

then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ *is strongly-consistent.*

Proof. Because $\langle M, \Phi_M\rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exists an $o : C$ in τ for which the contextAttribute violates ϕ. We set the value of contextAttribute such that it satisfies ϕ as shown in Figure 6.11. Now, $\tau' \models \phi$. Furthermore, no constraint in Φ_M is violated because contextAttribute is not related to another property (i), to itself (ii), and its value does not have to be unique (iii). Thus, $\tau' \models \Phi_M$, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent. □

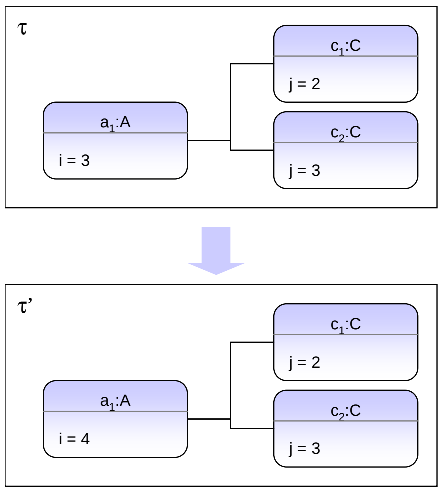


Figure 6.11: Illustration of the proof for Theorem 12.

The following constraints are inconsistent: Whereas the first invariant requires that for each object of class A, the value of attribute i must be less than the same attribute of all related A objects, the second invariant requires that the value of attribute i must be equal to the sum of the values of attribute i of all related A objects.

```
context A
   inv: AttributeRelation(b.c.a,i,>,i)
   inv: AttributeSumRestriction(i,b.c.a,i)
```

6.3.2.2 *Attribute Sum Restriction.*

The *Attribute Sum Restriction* pattern can be used limit the value of an integer attribute to the sum of related attributes. We defined the pattern as follows.

```
pattern AttributeSumRestriction(navigation: Sequence(Property),
                                summand: Property, summation: Property) =
 self.navigation.summand->sum() <= summation
```

We define the consistency theorem for the *Attribute Sum Restriction* pattern as follows.

Theorem 13. *Let* $\langle M, \Phi_M \rangle$ *be a strongly-consistent model,* $\phi(navigation, summand, summation)$ *be an instance of the* Attribute Sum Restriction *pattern with class* C *as context. If*

(i) there is no instance of the Attribute Relation, Attribute Value Restriction, Multiplicity Restriction, *or another* Attribute Sum Restriction *pattern in* Φ_M *in which* $summation$ *is used as parameter and*

(ii) if $navigation$ *is reflexive and* $summation = summand$*, there is no instance of the* Unique Identifier *pattern in* Φ_M *in which* $summation$ *is one of the unique properties,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ *is strongly-consistent.*

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exists an object $o : C$ for which the $summation$ attribute does not have the correct value. To this end, we set the value of $summation$ to the sum of the $summand$ properties of all objects related to o via $navigation$ as shown in Figure 6.12. Now, $\tau' \models \phi$. Further, the value of $summation$ is not related to any other property (i) and it is not related to itself (ii). Thus, no existing constraint in Φ_M is violated, $\tau' \models \Phi_M$, and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

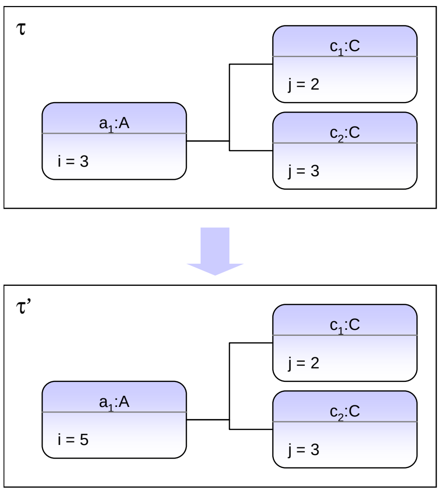


Figure 6.12: Illustration of the proof for Theorem 13.

The following constraints are inconsistent: Whereas the first invariant requires that the value of attribute i must be equal to the sum of the values of all i attributes of related A objects, which leaves zero as the only possible value for i for two or more related objects, the second invariant requires i to have a unique value for all objects of class A, which is a contradiction.

```
context A
  inv: AttributeSumRestriction(i,b.c.a,i)
  inv: UniqueIdentifier(i)
```

6.3.2.3 *Attribute Value Restriction.*

The *Attribute Value Restriction* pattern represents a common kind of constraint, namely simple value restrictions for attributes. We defined it as follows.

```
pattern AttributeValueRestriction (property:Property,operator,value:OclExpression) =
  self.property operator value
```

We define the consistency theorem for the *Attribute Value Restriction* pattern as follows.

Theorem 14. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, $\phi(p, op, v)$ be an instance of the* Attribute Value Restriction *pattern with class C as context. If*

(i) there is no instance of the Attribute Sum Restriction, Attribute Relation, Multiplicity Restriction, *or another* Attribute Value Restriction *pattern in Φ_M in which p is used as a parameter,*

(ii) if op is "=", there is no instance of the Unique Identifier *pattern in Φ_M in which p is one of the unique properties,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exists an $o : C$ in τ for which property p violates ϕ. We set the value of p such that it satisfies ϕ as shown in Figure 6.13. Now, $\tau' \models \phi$. Furthermore, no constraint in Φ_M is violated because p is not related to another property or to itself (i), and its value does not have to be unique (ii). Thus, $\tau' \models \Phi_M$, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

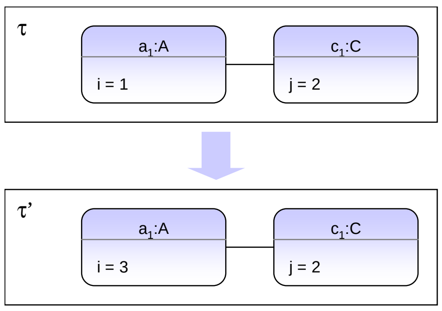


Figure 6.13: Illustration of the proof for Theorem 14.

The following constraints are obviously inconsistent.

```
context A
   inv: AttributeValueRestriction (i,<,0)
   inv: AttributeValueRestriction (i,>,0)
```

6.3.2.4 *Unique Identifier.*

Using the *Unique Identifier* pattern, a tuple of properties can be specified that have to be unique for each object of the context class. We defined it as follows.

```
pattern UniqueIdentifier(property:Tuple(Property)) =
  self.allInstances()->isUnique(property)
```

We define the consistency theorem for the *Unique Identifier* pattern as follows.

Theorem 15. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model, ϕ be an instance of the* Unique Identifier *pattern with class C as context, and a set P of properties. If for all $p \in P$,*

(i) the domain of the type of p is infinite,
(ii) there is no instance $\psi(p, \_, p)$ of the Attribute Sum Restriction *pattern in Φ_M,*
(iii) there is no instance $\psi(\_, \_, =, p)$ of the Attribute Relation *pattern on C in Φ_M, and*
(iv) there is no instance $\psi(p, =, \_)$ of the Attribute Value Restriction *pattern in Φ_M,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exist two objects o_1, o_2 of class C for which $p_i(o_1) = p_i(o_2)$ for all $p_i \in P$. We nondeterministically choose some i and change the value of $p_i(o_2)$ such that there is no other $o : C$ with $p_i(o) = p_i(o_2)$, as shown in Figure 6.14. Now, $\tau' \models \phi$. This is possible because there are infinitely many possible values for p_i (i). Furthermore, there would be only one possible value for p_i in the presence of a reflexive *Attribute Sum Restriction* constraint, which we exclude in the assumptions (ii) and objects of class C are not required to have the same value (iii, iv). After this change, it is possible that $\tau' \not\models \Phi_M$ if p_i is the parameter of an instance of the *Attribute Sum Restriction*, *Attribute Relation*, or *Attribute Value Restriction* pattern. In this case, the attribute values in τ' must be changed such that $\tau' \models \Phi_M$. This is possible because each attribute value is constrained by at most one constraint because of the assumption that $\langle M, \Phi_M \rangle$ is strongly-consistent and Theorems 13, 12, and 14. Now, $\tau' \models \Phi_M$ and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

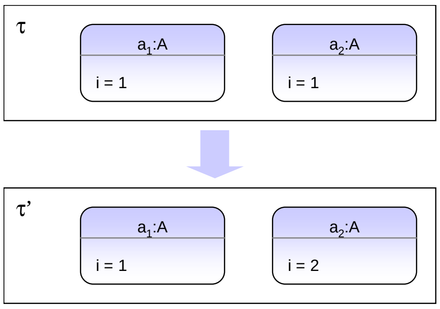


Figure 6.14: Illustration of the proof for Theorem 15.

The following constraints are inconsistent as explained above for the *Attribute Sum Restriction* pattern.

```
context A
   inv: UniqueIdentifier(i)
   inv: AttributeSumRestriction(i,b.c.a,i)
```

6.3.3 Other Patterns.

There is only one pattern that restricts both the structure of the object graph and attribute values, *Multiplicity Restriction*.

6.3.3.1 *Multiplicity Restriction.*

The *Multiplicity Restriction* pattern can be instantiated to limit the multiplicity of an association to a given attribute value. We defined it as follows.

```
pattern MultiplicityRestriction (navigation: Sequence(Property),
                                 operator: OclExpression, value:OclExpression) =
 self.navigation->asSet()->size() operator value
```

Since the *Multiplicity Restriction* pattern restricts both the structure of the object graph and attribute values, it is related to almost all other patterns as shown in the previous theorems. We take this into account in the consistency theorem, which we define for the *Multiplicity Restriction* pattern as follows.

Theorem 16. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(P, op, v)$ be an instance of the* Multiplicity Restriction *pattern with class C as context. If*

(i) *v is not a property that is used as parameter for an instance of the* Attribute Sum Restriction, Attribute Relation, Attribute Value Restriction, *or another* Multiplicity Restriction *pattern in Φ_M,*

(ii) *the tuple (op, v) is not one of the following: $(<, 1), (=, 0), (\leq, 0)$ while the lower multiplicity bound of the last element of P is greater than zero,*

(iii) *for each $p \in P$, the multiplicity is unbounded (*),*

(iv) *there is no other instance of the* Multiplicity Restriction *pattern on any $p \in P$ in Φ_M, and*

(v) *p is not used as part of a parameter value for any instance of the* No Cyclic Dependency, Object In Collection, Surjective Association, Injective Association, *or* Type Relation *pattern in Φ_M, and p^{-1} is not used in any instance of the* Type Restriction *pattern in Φ_M,*

then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. Because $\langle M, \Phi_M \rangle$ is strongly-consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that serves as a witness for the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models \phi$.

In this case, there exists an object $o : C$ that has either too few or too many links to objects of class $type(P)$. To this end, we create or delete objects of class $type(P)$ until $\tau' \models \phi$ as shown in Figure 6.15. Creating or deleting such objects does not violate any constraint in Φ_M because P is not constrained by any other constraint (v); furthermore, at least one object of class $type(P)$ will remain because the case is excluded that zero objects of class $type(P)$ are connected to o. This is because of the following reasons: The value of v is not constrained if v is a property (i), the multiplicity of P is unbounded (iii), there is no other instance of *Multiplicity Restriction* on p (iv), and the parameters of ϕ allow for at least one link (ii). Thus, $\tau' \models \Phi_M$, and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent. □

The following constraints are inconsistent: Whereas the first invariant limits the number of associations between an A object and a B object to the value of i, the second invariant determines this value to be zero. The third invariant contradicts the previous two by requiring at least one B object to be related to each A object.

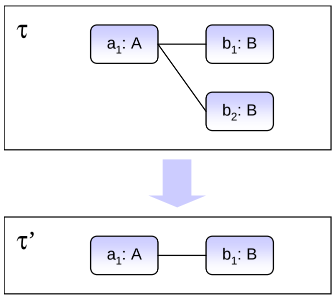


Figure 6.15: Illustration of the proof for Theorem 16.

```
context A
   inv:  MultiplicityRestriction (b,<=,i)
   inv: AttributeValueRestriction ( i ,=,0)
   inv:  MultiplicityRestriction (b,>=,1)
```

6.3.4 Summary of Constraint Pattern Dependencies.

We have analyzed the dependencies between the elementary constraint patterns from the constraint pattern library defined in Section 4.3. This allows for deciding whether pattern instances used for refinement are consistent or whether further consistency analysis is required. Figure 6.16 summarizes the possible conflicts between instances of constraint patterns.

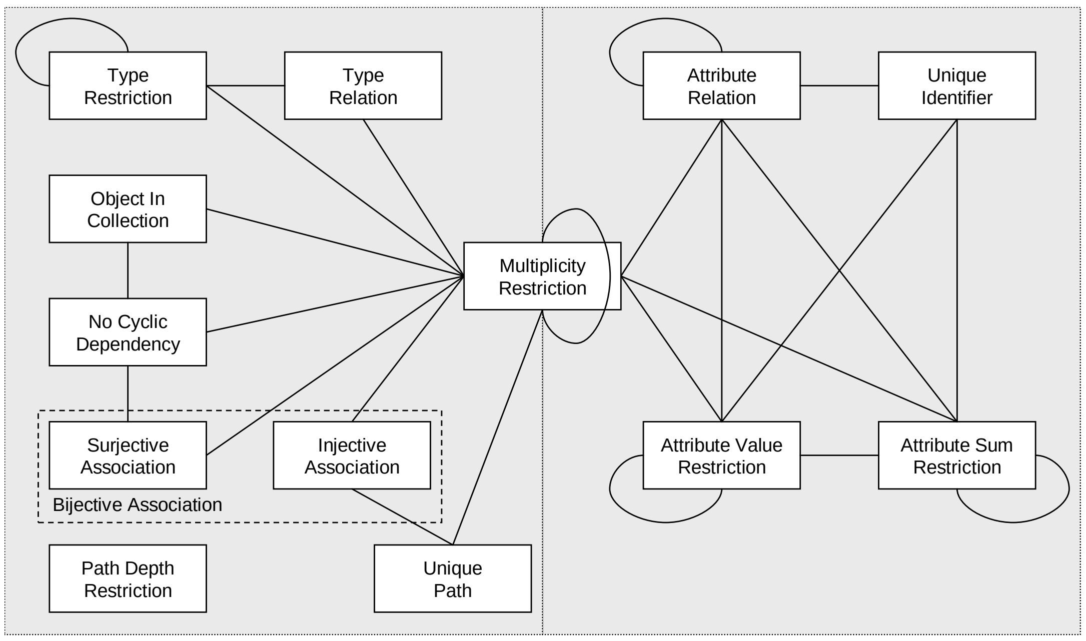


Figure 6.16: Possible conflicts between instances of constraint patterns.

The patterns in Figure 6.16 are divided into two parts. Patterns that constrain associations are in the left half, patterns that constrain attribute values are in the right half, and the *Multiplicity Restriction* pattern belongs to both halves. Interestingly, there is no direct dependency between patterns in the left part and patterns in the right part. This means that an instance of a pattern that restricts an association can never contradict an instance of a pattern that restricts the value of an attribute. However, they can contradict each other as soon as the *Multiplicity Restriction* pattern is instantiated. This motivates the following corollary.

Corollary 1. *Two pattern instances $\phi_1, \phi_2 \in \Phi$ can be inconsistent iff there exists a path between them in the graph in Figure 6.16 and there exists a pattern instance in Φ for each node on this path.*

Having introduced consistency theorems for the elementary patterns in our library, we discuss the consistency of composite patterns next.

6.4 Consistency of Composite Constraint Patterns

In Section 6.3, we have discussed the relations between *elementary* patterns only. However, in Chapter 4, we have defined the concept of *composite* constraint patterns, which can be used to logically connect pattern instances. In the following, we discuss how the consistency of composite constraint patterns can analyzed. Note that with the current set of consistency theorems, all composite patterns can be analyzed except for *Negation*.

Negation.

In our consistency observations in Section 6.3, we assume that all pattern instances are not negated. The reason is that, among all composite patterns, this pattern has the strongest consequences for the consistency of constraint specifications: The consistency theorems as stated previously often do not hold anymore if negation is used. If negated pattern instances occur in a constraint specification, our analysis currently warns the user that the respective pattern instance cannot be analyzed. The user must subsequently decide whether the warning is a false positive or an actual inconsistency.

In order to support negation, the set of consistency theorems would need to be extended by one new consistency theorem for every negated pattern. The effort for such an extension varies between the constraint patterns: Whereas the consistency theorems of some patterns would still hold if the respective pattern instances were negated, e. g., of the *Attribute Value Restriction* pattern ($\neg(x < y) \Leftrightarrow x \geq y$), the negation of other patterns can introduce new inconsistencies. For example, a negated instance of the *No Cyclic Dependency* pattern can contradict an instance of the *Path Depth Restriction* pattern.

An alternative approach would be to convert pattern instances into a normal form in which negation does not occur. For example, the following pattern instances are equivalent.

```
context Office
inv:  MultiplicityRestriction (inhabitant,<=,desks)
inv: Negation( MultiplicityRestriction (inhabitant,>,desks))
```

Such transformation into normal form may require the introduction of new patterns. For example, the pattern library must be complemented with an inverse of the *No Cyclic Dependency* pattern, i. e., a pattern that *requires* cycles to occur in model states. We consider developing such a normal form an interesting direction for future work.

Conjunction and Disjunction.

Conjunctions of pattern instances can be expressed using the *And* pattern. Pattern instances that are part of a conjunction do not have to be treated specially in consistency analysis because each conjunct must hold such that the conjunction holds. In our analysis, we thus treat each conjunct as an elementary constraint that must be consistent with all other constraints. Therefore, we establish the following consistency theorem.

Theorem 17. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(P)$ be an instance of the* And *pattern with a list $P = p_1.\dots.p_n$ of pattern instances. If $\langle M, \Phi_M \cup \{p_1, \dots, p_n\} \rangle$ is strongly-consistent, then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.*

Proof. The theorem holds because the implication $(\Phi \vdash P$ and $\Phi \vdash Q) \Rightarrow (\Phi \vdash P \wedge Q)$ is a rule of the sequent calculus for first-order logic [Gallier, 1986]. □

Disjunctions of pattern instances can be expressed using the *Or* pattern. If a pattern instance is part of a disjunction and its consistency assumptions do not hold, the disjunction can still be consistent if there is another pattern instance in the same conjunction for which the consistency assumptions hold because of the equivalence $\bigvee P_i \Leftrightarrow \exists i.P_i$. We establish the following consistency theorem for the *Or* pattern.

Theorem 18. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(P)$ be an instance of the* Or *pattern with a list $P = p_1.\dots.p_n$ of pattern instances. If there exists a $p_i \in P$ such that $\langle M, \Phi_M \cup \{p_i\} \rangle$ is strongly-consistent, then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.*

Proof. The theorem holds because the implication $(\Phi \vdash P$ or $\Phi \vdash Q) \Rightarrow (\Phi \vdash P \vee Q)$ is a rule of the sequent calculus for first-order logic [Gallier, 1986]. □

Implication.

The *If-Then-Else* pattern can be used to model implication between pattern instances. A formula (if I then T else E) is equivalent to the expression $(\neg I \vee T) \wedge (I \vee E)$. Since our analysis does not handle negated constraints, we must state stronger assumptions in the consistency theorem for the *If-Then-Else* pattern: If both T and E are consistent, then the instance of the *If-Then-Else* pattern is consistent because $T \wedge E \Rightarrow (\neg I \vee T) \wedge (I \vee E)$. Thus, $T \wedge E$ is a sufficient, but not necessary condition for the consistency of the *If-Then-Else* statement. We establish the following consistency theorem.

Theorem 19. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(I, T, E)$ be an instance of the* If-Then-Else *pattern with I, T, E being lists of pattern instances. If*
(i) $\langle M, \Phi_M \cup \{\bigwedge T\} \rangle$ is strongly-consistent and
(ii) $\langle M, \Phi_M \cup \{\bigwedge E\} \rangle$ is strongly-consistent,
then, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly-consistent.

Proof. The semantics of *If-Then-Else* (I,T,E) is $I \Rightarrow T \wedge \neg I \Rightarrow E$, which is equivalent to $(\neg I \vee T) \wedge (I \vee E)$. Thus, if T and E are consistent, *If-Then-Else* (I,T,E) is consistent. □

Note that the consistency observations in Theorem 19 are independent of the assumptions I.

Quantification.

Using the *Exists* and *ForAll* patterns, pattern instances can be limited such that they do not need to hold for all instances of a certain class, but only for a subset. For both patterns, the pattern instances used in their parameters must be consistent such that the quantified expression is consistent. The consistency theorem for the *ForAll* pattern reads as follows.

Theorem 20. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(P, S)$ be an instance of the* ForAll *pattern with P being a list of pattern instances and S being a set of objects. If $\langle M, \Phi_M \cup \{\bigwedge P\}\rangle$ is strongly-consistent, then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent.*

The consistency theorem for the *Exists* pattern reads as follows.

Theorem 21. *Let $\langle M, \Phi_M \rangle$ be a strongly-consistent model and $\phi(P, S)$ be an instance of the* Exists *pattern with P being a list of pattern instances and S being a set of objects. If $\langle M, \Phi_M \cup \{\bigwedge P\}\rangle$ is strongly-consistent, then, $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly-consistent.*

For both patterns, the assumptions under which instances of the respective patterns are consistent are *sufficient* but not *necessary* because the pattern instances P must only hold for elements in S. However, our assumptions are independent of S and thus, more strict. This simplifies consistency analysis and the following proof, which is applicable to both theorems.

Proof. If $P(x)$ is consistent, both $\forall x.P(x)$ and $\exists x.P(x)$ are consistent. Formally, $P(x) \Rightarrow \forall y.P(y)$ and $P(x) \Rightarrow \exists y.P(y)$ [Gallier, 1986]. □

6.5 Discussion

We have introduced an approach for analyzing the consistency of pattern-based constraint specifications. In addition, we have stated and proven a consistency theorem for each constraint pattern in our pattern library.

In this section, we discuss the theoretical and practical limitations and implications of our approach. In particular, we illustrate the limitations of our approach (Section 6.5.1) and evaluate our approach in comparison to other analysis approaches (Section 6.5.2).

6.5.1 Limitations of the Approach.

In this subsection, we point out four limitations of our analysis approach: arbitrary OCL constraints cannot be analyzed, return values of methods cannot be used as parameter values in patterns, adding new constraint patterns requires substantial effort, and providing sufficient conditions in the consistency theorems generates false positives.

Arbitrary OCL Constraints Our analysis only analyzes the consistency of constraints that are instances of constraint patterns. Thus, the consistency of models that are additionally annotated with arbitrary OCL constraints, which includes instances of the *Literal OCL* pattern, cannot be analyzed.

We see two choices if arbitrary OCL constraints appear in a constraint specification. First, each OCL constraint can be abstracted into a constraint pattern and a consistency theorem for this new pattern can be established. However, this is often not feasible because adding new constraint patterns to a pattern library requires significant effort in updating the existing consistency theorems. Second, the development process can be split into two phases. In the first phase, the model is only augmented with pattern instances and subsequently analyzed for consistency. This allows model developers to receive immediate feedback about the consistency of one part of the constraint specification because of the polynomial complexity of our approach. In the second phase, the model is augmented with the remaining constraints in OCL. The model developer must then analyze the consistency of the fully constrained model using one of the analysis approaches introduced in Section 6.1, which typically consumes more time than our approach.

Method Calls in Pattern Instances In our pattern library from Section 4.3, the types of the pattern parameter are typically properties or sets thereof, classes, or integer. In typical constraint languages, it is also possible to call methods in invariants. For example, the fact that managers should not be able to hire themselves could be expressed as an invariant as not self.hire(self) – although such a condition should clearly be stated in the pre-condition of hire(). When analyzing the consistency of this invariant, the definition of hire() must be taken into account. Since methods can be recursive and potentially nonterminating, which further complicates analysis, we do not allow methods as parameter types for our patterns, following [Darvas and Müller, 2006].

Adding New Constraint Patterns In Chapter 4, we have emphasized that the library of constraint patterns presented is extensible. However, when the library gets extended, two tasks must be carried out with respect to consistency. First, a new consistency theorem for the newly added constraint pattern must be established. Second, the consistency theorems of all existing patterns must be revised. In particular, for each theorem, it must be analyzed whether instances of the newly added pattern can cause inconsistencies with instances of the respective pattern. Not all consistency theorems must be changed though: If the newly added pattern is a pure attribute-restricting pattern, it will not contradict pure association-restricting patterns and vice versa.

False Positives Since the assumptions in our consistency theorems represent *sufficient* conditions and not *necessary* conditions, there are models that are consistent, but our analysis cannot determine the consistency because the assumptions are overly restrictive. Consider the following example.

```
context Manager
inv: TypeRelation(worksIn,{Single})
inv:  MultiplicityRestriction (worksIn,=,1)
```

These constraints are consistent because they are satisfied by a model state in which an instance of Manager works in an instance of a Single office. However, our analysis cannot determine this consistency because assumption ii of the *Type Restriction* pattern and, by symmetry, assumption v of the *Multiplicity Restriction* pattern are violated. Thus, our analysis creates *false positives*, i. e., it warns of potential inconsistencies when the constraints are actually consistent. The amount of false positives can be decreased by weakening the assumptions in the consistency theorems. We consider this future work and discuss it in Section 9.2.

6.5.2 Evaluation of our Approach.

In this subsection, we evaluate our approach. First, we report on the fundamental properties of our approach: completeness, correctness, and complexity.

Completeness.

A consistency analysis is complete if for each UML/OCL model, the analysis returns *true* if the model is consistent and *false* otherwise. Our approach is a heuristic approach. As a consequence, it returns *true* if the model can be shown consistent and *don't know* otherwise. Thus, our approach is **incomplete**.

Correctness.

A consistency analysis is correct if it returns *true* for each consistent UML/OCL model and does not return *true* for an inconsistent model. Since our approach only returns *true* if the consistency assumptions, which have been proven, hold, and does not return anything otherwise, it is **correct**.

Complexity.

The complexity of a consistency analysis is the computational effort that is necessary to compute the result. In our approach, the consistency assumptions for each pattern instance must be checked. This involves searching for pattern instances in the constraint specification Φ that can violate the assumptions. Many consistency theorems contain assumptions about a given path P, for which each element must be analyzed. Thus, checking the assumptions for one consistency theorem is $\mathcal{O}(|\Phi| \cdot |P|)$ in the worst case, where $|\Phi|$ is the number of pattern instances in the constraint specification and $|P|$ is the length of the longest path used as parameter value. The analysis of a whole constraint specification Φ is thus in $\mathcal{O}(|\Phi|^2 \cdot |P|)$.

Next, we evaluate our approach according to the criteria defined in Section 6.1. Our approach covers all elements of UML and the subset of OCL that is expressible using constraint patterns. The hypothesis of our approach is that the constraint specification is consistent. Our approach is flexible with respect to different notions of consistency because different notions of consistency can be supported by stating and proving consistency theorems for the respective consistency notion.

The input to our approach is the model and its constraint specification as a set of pattern instances. The output is “Yes” in case that all pattern instances satisfy the consistency assumptions of their respective pattern definition or “Don't know” otherwise. In the latter case, a secondary analysis is required, which we discuss this in detail in Section 6.6. Whereas stating and proving the consistency theorems is manual, the actual consistency analysis is fully automated and can be done in polynomial time ($\mathcal{O}(|\Phi|^2 \cdot |P|)$).

In Table 6.1, we give an overview of all analysis approaches discussed in this chapter, including our novel approach based on constraint patterns.

As explained in Section 6.2, we have developed our approach with focus on efficiency and automation. Besides SAT-based approaches, our approach is the only one that offers full automation to model developers. In contrast to SAT-based approaches, our approach has polynomial complexity, which allows consistency analysis also for large models. In Chapter 8, we perform case studies using our approach and report on practical experiences.

	Interactive Theorem Proving	Witness Creation	SAT and Model Checking	Pattern-Based Heuristics
coverage	Almost full UML/OCL	Full UML/OCL	UML, subset of OCL	UML, subset of OCL
hypothesis	exchangeable	“consistent”	“consistent”	“consistent”
flexibility	high	high	low	medium
input	model, constraints, proof obligations, proof	model, constraints, witness	model, constraints	model, pattern-based constraints
output	done or oops	yes or don’t know	yes or don’t know (divergence, out-of-memory)	yes or don’t know
failure	use weaker obligation	no information	increase search space	secondary analysis
complexity	undecidable	undecidable	exponential	polynomial
automation	little	little	high	high

Table 6.1: Comparison of consistency analysis approaches.

6.6 A Consistency-Aware MDE Process

In this section, we show how our analysis approach integrates into an MDE process. Figure 6.17 shows such a process as a workflow model. The first and the last task in this process, *specify class model* and *generate code*, are the same as in common MDE processes. Compared to common MDE processes, we introduce one new task, *refine model*, extend one common task, *analyze consistency*, and change the control flow with two new decisions c_1 and c_2.

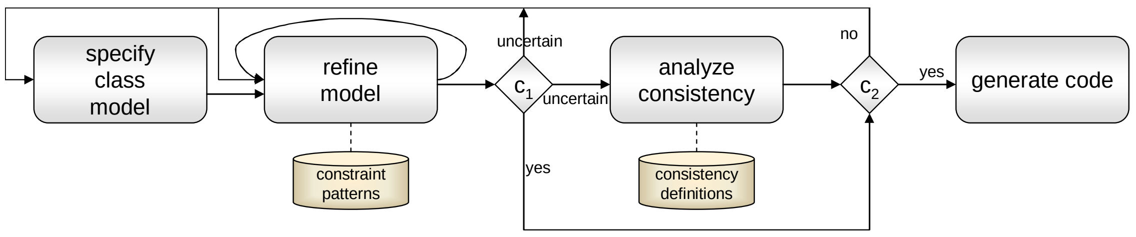


Figure 6.17: A consistency-aware MDE process.

The new task, *refine model*, is based on constraint patterns stored in a repository. This repository also stores the consistency theorem for each pattern, which describe the assumptions under which instances of the respective pattern preserve the consistency of the model. Refining the model is an iterative task and thus displayed with a loop in the figure. In each refinement step, the model developer chooses a constraint pattern x from the repository and instantiates it. Upon instantiation, the assumptions in the consistency theorem for x are checked.

After the model has been refined, the control flow splits at c_1. If during refinement, each pattern instance has satisfied the consistency assumptions of the respective constraint pattern, the model consistent and the control flow proceeds to *generate code*. If the consistency assumptions of one or more refinement steps do not hold, the consistency of the whole specification is uncertain. In this case, the model developer has two choices. First, he can go back in the process and either adjust the class model or the constraint specification. Second, a secondary consistency analysis can be carried out in the task *analyze consistency* using the approaches and tools introduced in Section 6.1. The control flow splits again at c_2: If after the secondary analysis the constraint specification is still inconsistent, control flow can go back either to *specify class model* where the model can be changed or to *refine model* where the pattern instances can be revised.

If the constraint specification could be shown consistent, control flow goes to *generate code*. In this task, the constraint-pattern instances are automatically transformed into sentences in a formal specification language, e. g., OCL. Subsequently, the generated specification can be used to validate model states against the constraints.

To support users in the step “refine model,” we have implemented an extension to the CASE tool IBM Rational Software Architect (RSA) that adds a repository of constraint patterns to the tool. When a constraint pattern is instantiated and parametrized, the tool analyzes whether the consistency assumptions of the respective pattern hold. We elaborate on tool support in Chapter 7.

6.7 Summary

In this chapter, we have introduced an approach to consistency analysis of class models constrained by pattern-based constraint specifications. To this end, we have stated and proven for each elementary constraint pattern a theorem that defines sufficient conditions for the consistency of the pattern’s instances. Furthermore, we have shown that it can be decided in polynomial time whether the consistency assumptions for a given pattern instance hold. Our approach complements existing approaches for the consistency analysis of UML/OCL by providing an efficient way of heuristically determining consistency and thus, it can be effectively used after each single refinement step.

Chapter 7

Tool Support

In the previous chapter, we have introduced an approach to developing consistent constraint specifications based on constraint patterns. Since the success of MDE strongly depends on tool support, we have implemented a set of plug-ins for the MDE tool IBM Rational Software Architect (RSA) that enables consistency-preserving refinement of UML class models with constraint patterns.

We call this set of plug-ins COPACABANA, and we describe them in detail in this chapter. COPACABANA supports model developers in the four phases of our approach: constraint elicitation, constraint specification, consistency analysis, and code generation of constraints in class models.

This chapter is structured as follows. We first give a brief overview of RSA and its Application Programming Interface (API). In Section 7.2, we give an overview of the components of COPACABANA and how they integrate in RSA. We summarize this chapter in Section 7.3 and elaborate on details of the implementation in Appendix B.

7.1 A Short Introduction to IBM Rational Software Architect

Our tool support builds on the MDE tool IBM Rational Software Architect (RSA), which itself builds on the generic development platform Eclipse [Eclipse Foundation, 2007b]. Choosing a "commercial off-the-shelf" tool as basis for our tool support has several advantages. First, building on an existing tool increases the acceptance among target users. Second, it speeds up development of our tools because RSA provides various frameworks, which offer APIs for extensibility. In particular, we extend the following frameworks.

Modeling The modeling framework provides support for modeling UML models and graphically representing them. In particular, its Graphical User Interface (GUI) allows model developers to edit models by graphical means, e. g., drag-and-drop. In addition, the modeling framework comprises an OCL parser that checks OCL expressions for syntactic correctness and type conformance.

Patterns The patterns framework adds support for any kind of patterns. It provides a graphical editor for defining patterns and their signatures in a model-driven way. The pattern semantics can be defined in Java by specifying code that is executed when patterns are instantiated and parameterized. All available patterns are shown to the model developer in a special view, the *pattern explorer*. From this view, patterns can be chosen and instantiated. As shown in Section 2.4, pattern instances are represented as UML collaborations in RSA.

Validation The validation framework provides a uniform interface to model validation rules to both rule developers and rule users. From the user's point of view, models and their elements can be validated against available validation rules from the models' context menus. The validation results are uniformly displayed in the *problems* view of RSA. From the developer's point of view, new validation rules can be added to the system using the API of the framework. The rules can be evaluated either in "*live*" mode when a model is changed or in "*batch*" mode by explicit invocation of the validation framework.

Transformations The transformations framework provides a uniform interface to model transformations to both transformation developers and transformation users. From the user's point of view, all available transformations are displayed in a joint view and can be invoked from there after a transformation *configuration* has been created. Transformation configurations specify the parameters for the transformations such as the input model and the output model. From the developer's point of view, transformations can be specified as rules for certain kinds of model elements. When the user invokes a transformation on a given model, the transformation framework iterates over all elements in the model and invokes the appropriate transformation rules.

7.2 Overview of COPACABANA

COPACABANA supports users in developing concise and consistent constraint specifications for class models. The tool is based on and extends IBM Rational Software Architect (RSA) via a plug-in mechanism. Figure 7.1 provides an overview of the architecture of our solution. The top row shows the different components of our approach. The vertical axis shows how the components build on the frameworks of RSA. For example, the *constraint elicitation* component builds on the modeling framework of RSA. Each framework of RSA builds on the Eclipse platform.

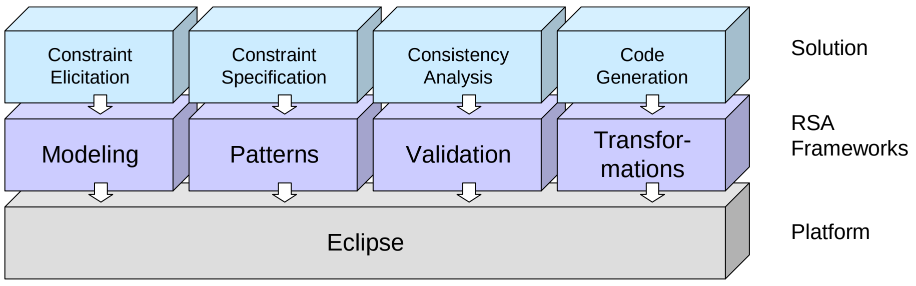
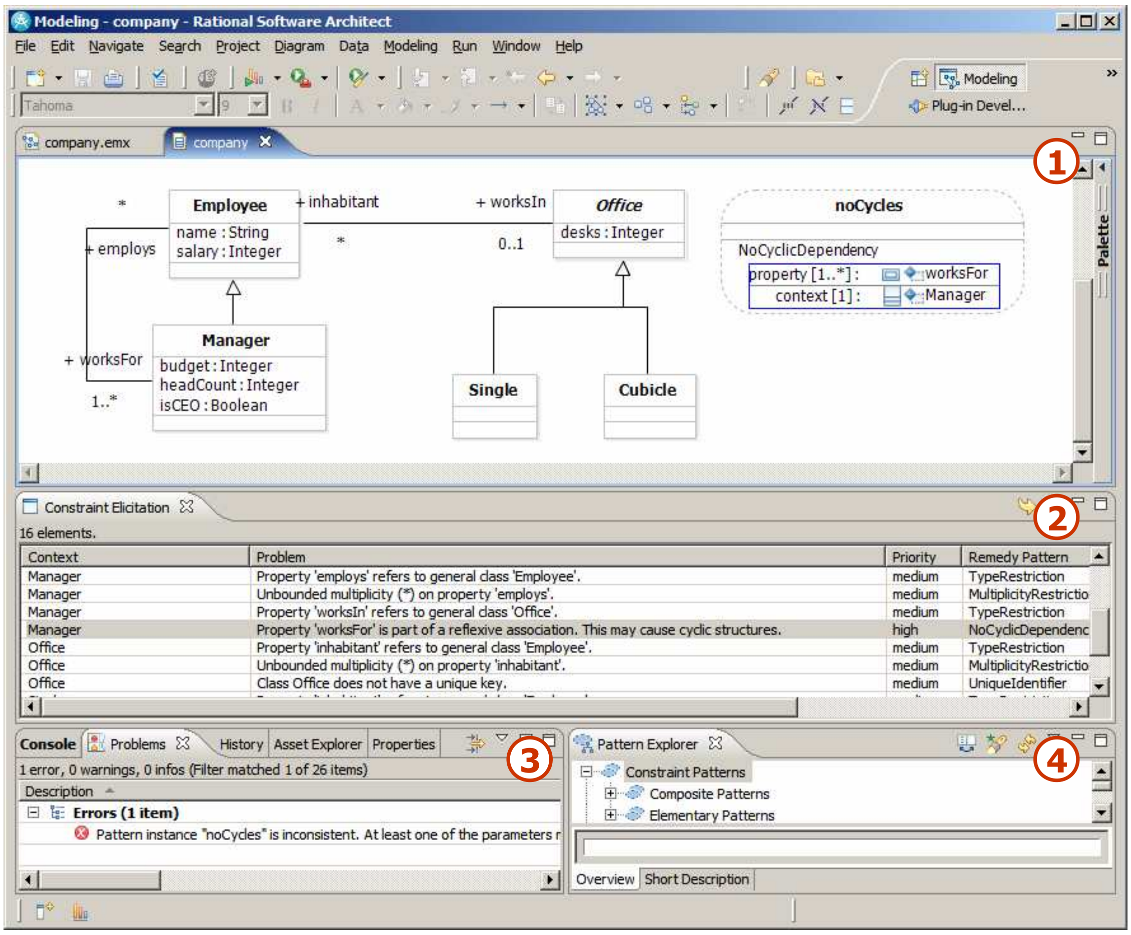


Figure 7.1: The architecture of COPACABANA.

Figure 7.2 shows a screenshot of RSA with the COPACABANA plug-ins. The largest view (1) contains the company model and a collaboration that represents an instance of the *No Cyclic Dependency* pattern. In this view, models can be edited and pattern instances be parametrized via drag-and-drop. Underneath, the *constraint elicitation* view (2) shows the results of the constraint elicitation component.

The bottom part of the window contains two more views. First, the results of the consistency analysis are shown in the bottom left part in the *problems* view (3) of RSA. The last view in this figure is the above-mentioned *pattern explorer* (4), which displays



16 elements.

Context	Problem	Priority	Remedy Pattern
Manager	Property 'employs' refers to general class 'Employee'.	medium	TypeRestriction
Manager	Unbounded multiplicity (*) on property 'employs'.	medium	MultiplicityRestrictio
Manager	Property 'worksIn' refers to general class 'Office'.	medium	TypeRestriction
Manager	Property 'worksFor' is part of a reflexive association. This may cause cyclic structures.	high	NoCyclicDependenc
Office	Property 'inhabitant' refers to general class 'Employee'.	medium	TypeRestriction
Office	Unbounded multiplicity (*) on property 'inhabitant'.	medium	MultiplicityRestrictio
Office	Class Office does not have a unique key.	medium	UniqueIdentifier

1 error, 0 warnings, 0 infos (Filter matched 1 of 26 items)

Description

Errors (1 item)

Pattern instance "noCycles" is inconsistent. At least one of the parameters r

Figure 7.2: Screenshot of the COPACABANA prototype in RSA.

available patterns along with a description. The pattern explorer furthermore displays all available transformations in RSA's transformation framework.

In the following, we provide detailed explanations about each component shown in Figure 7.1. To this end, we apply our approach to the company model.

7.2.1 Constraint Elicitation.

This component searches class models for occurrences of the anti-patterns as defined in Chapter 3 and complements domain analysis as explained in the same chapter. These anti-patterns comprise model elements that are typically specified at a high level of abstraction and thus typically require refinement with textual constraints. With COPACABANA, *constraint elicitation* can be invoked from the context menu of a model as shown in Figure 7.3.

The constraint elicitation component displays a view in which the analysis results are presented in a table. Each result comprises four parts: a *context*, i. e., the class that matches an anti-pattern, a *description* of the anti-pattern that this class is a part of, the *priority* of the respective anti-pattern as defined in Section 3.2, and a list of constraint patterns that can be used to *remedy* the respective anti-pattern. The entries in the table can be sorted by either the context class or the priority by clicking on the header of the corresponding table column. Figure 7.4 shows the constraint elicitation view for the company model.

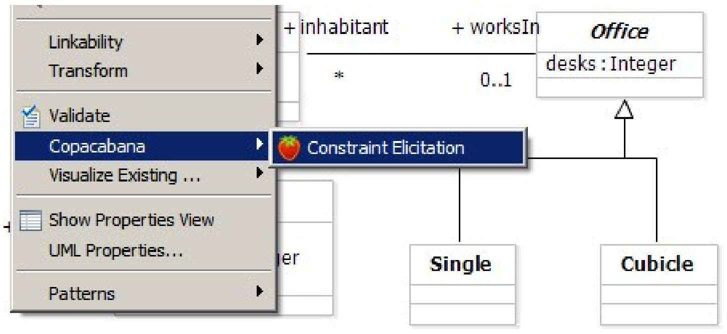
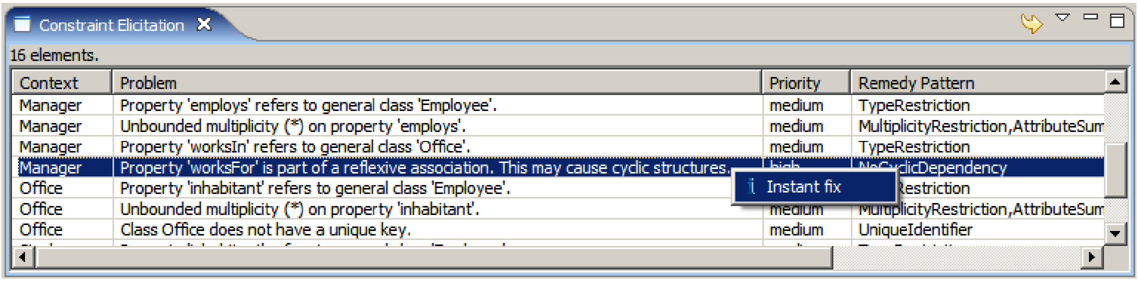


Figure 7.3: Invoking constraint elicitation.



16 elements.

Context	Problem	Priority	Remedy Pattern
Manager	Property 'employs' refers to general class 'Employee'.	medium	TypeRestriction
Manager	Unbounded multiplicity (*) on property 'employs'.	medium	MultiplicityRestriction,AttributeSum
Manager	Property 'worksIn' refers to general class 'Office'.	medium	TypeRestriction
Manager	Property 'worksFor' is part of a reflexive association. This may cause cyclic structures.	high	NoCyclicDependency
Office	Property 'inhabitant' refers to general class 'Employee'.		Restriction
Office	Unbounded multiplicity (*) on property 'inhabitant'.	medium	MultiplicityRestriction,AttributeSum
Office	Class Office does not have a unique key.	medium	UniqueIdentifier

Figure 7.4: Results of constraint elicitation.

As we pointed out in Section 3.3, not all results represent actual problems. However, the model developer is advised to browse the list, identify actual problems, and react accordingly. For immediate reaction, each analysis result offers a context menu from which a solution for the respective problem can be automatically instantiated in the form of constraint patterns suitable for the respective result. From the context menu of each analysis result, an appropriate constraint pattern can be selected as *instant fix* and automatically be instantiated. This involves the *constraint pattern library*, as explained in the following subsection.

Such tool support has the following advantages. First, the user is supported in detecting anti-patterns, which is usually time-consuming, requires a high level of expertise from the model developer, and some anti-patterns may not be detected by the model developer, which may cause problems in the remainder of the development process. Second, the model developer can specify most constraints by simply instantiating and parameterizing constraint patterns instead of manually writing OCL expressions, which is time-consuming and error-prone because some constraints are fairly complicated, e. g., constraints for reflexive associations (cf. Section 3.1.3).

7.2.2 Constraint Specification.

COPACABANA provides an implementation of the library of composable constraint patterns introduced in Chapter 4. The implementation of constraint patterns builds on RSA's patterns framework as illustrated in Figure 7.1. Constraint patterns can be instantiated in two ways. First, they can be manually selected from RSA's pattern explorer and dragged-and-dropped onto a class model. Second, they can be automatically instantiated from the context menu of each analysis result from the constraint elicitation view.

When the model developer chooses to *manually* instantiate a certain constraint pattern, he selects the pattern from the pattern explorer view and creates an instance of the pattern by dragging-and-dropping it onto the class diagram. Subsequently, he can set the values for the parameters of the pattern instance. The values for parameters of nonprimitive types, e. g., Class or Property, can be set by dragging-and-dropping the appropriate model elements into the parameter slots of the graphical representation of the pattern instance. For example, the parameter *property* of the *No Cyclic Dependency* pattern can be set this way, as shown in Figure 7.2 (1). Values for primitive types, e. g., Integer or String, can be entered as text.

The model developer can also *automatically* instantiate constraint patterns from the constraint elicitation view. This way, the values for those parameters that are known from the constraint elicitation are automatically set. For example, the *Unrestricted Reflexive Associations* anti-pattern is detected for the class Manager and its property workFor. The model developer is suggested to instantiate the *No Cyclic Dependency* pattern, and if he does, both parameters of the pattern, *context* and *property*, can be set. For other patterns, not all parameters can be automatically set; the remaining parameters can be specified manually as described above.

Since constraint patterns are represented as UML collaborations, the parameters of *composite* patterns are of type Collaboration, which enables drag-and-drop composition of constraint patterns in the GUI. Figure 7.5 shows an example of using composite constraint patterns in our approach.

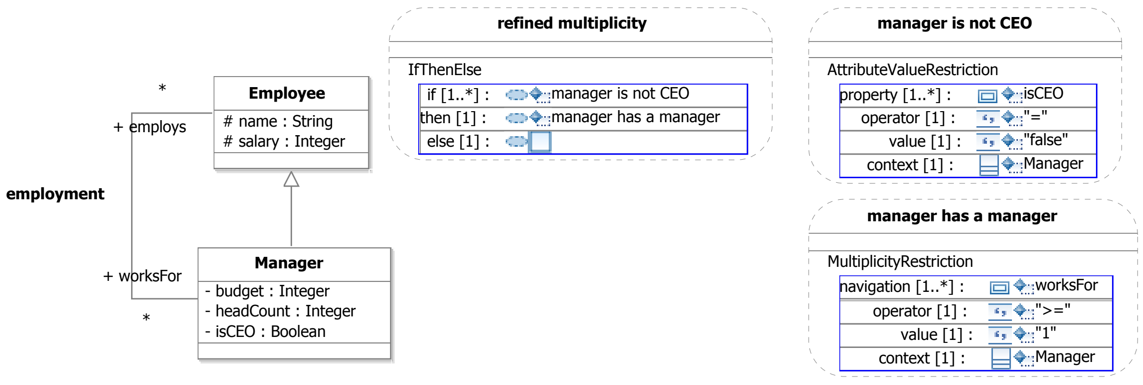


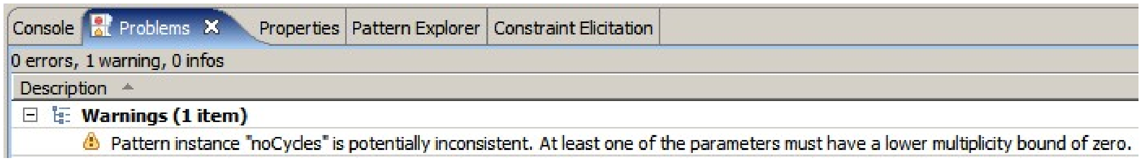
Figure 7.5: Example instance of a composite pattern.

7.2.3 Consistency Analysis.

COPACABANA provides an implementation of the pattern-based consistency analysis introduced in Chapter 6. To this end, we store with each pattern the assumptions from the respective consistency theorem, i. e., the assumptions under which the pattern can be instantiated in a consistency-preserving way.

The analysis uses these consistency assumptions and checks for each pattern instance whether the assumptions hold or not. If they hold, the constraint specification is consistent. If they do not hold, no statement about consistency can be made, as explained in Chapter 6. In this case, the consistency analysis component issues a *warning* that the pattern instance is *potentially* inconsistent, i. e., the consistency assumptions do not hold.

Figure 7.6 shows the analysis results for the company model and its constraints that we have added throughout this thesis. It shows a warning that constraint *noCycles* cannot be shown consistent. As explained in Section 5.1.1.3, this is caused by the fact that *noCycles* disallows cyclic management hierarchies, whereas the company model requires every manager to have at least one superior manager. As a result, there is no state of the company model with a nonempty, finite set of managers.



0 errors, 1 warning, 0 infos

Warnings (1 item)

Pattern instance "noCycles" is potentially inconsistent. At least one of the parameters must have a lower multiplicity bound of zero.

Figure 7.6: Consistency analysis results of the constrained company model.

In the case that consistency cannot be shown, the model developer must examine the warning. This can be either done by a secondary consistency analysis as explained in Section 6.6 or by a manual examination of the model and its constraint specification. Subsequently, the model developer can either adapt the model or correct the wrongly specified constraint if the warning represents an actual error.

In the case of the *noCycles* constraint, we decide to change the model to establish consistency. In order to satisfy the consistency assumptions for the *No Cyclic Dependency* pattern, we change the multiplicity of the worksFor association end from 1..$*$ to $*$, which makes it optional for employees to have superior managers. Now, the model is consistent because companies can be modeled with noncyclic management hierarchies in which the top manager does not have a superior manager.

7.2.4 Code Generation.

COPACABANA supports model developers in the fourth phase of our method by providing a model transformation that generates OCL code from pattern instances. This transformation can be found together with all other available transformations in the pattern explorer.

Figure 7.7 shows a screenshot of the pattern explorer with the context menu of our transformation. As explained in Section 7.1, model developers can create an new transformation configuration for the model under development and subsequently run the transformation.

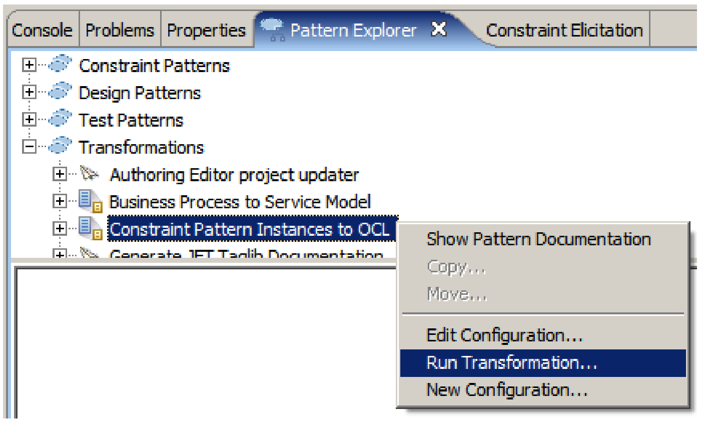


Figure 7.7: Invoking code generation in the pattern explorer view.

The result of the transformation is shown in Figure 7.8 and comprises two parts. First, the transformation has added an OCL constraint to the model, which is attached as an invariant to class Manager. Second, this invariant invokes an operation that computes the transitive closure of the worksFor association, as explained in Section 3.1.3. This operation has been added to class Manager and its body is defined declaratively in OCL. Note that in Figure 7.8, we have also adapted the multiplicity of the worksFor association end to make the model consistent as described in the previous subsection.

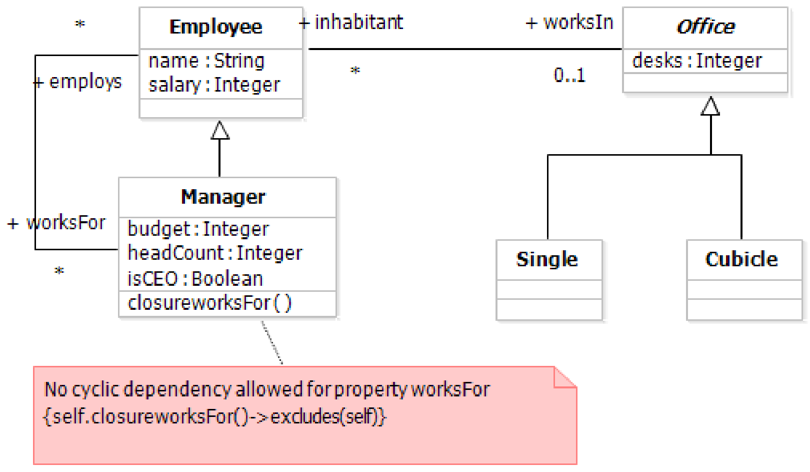


Figure 7.8: Company model after transformation of *noCycles* constraint.

We have used COPACABANA to refine the company model and went through the four phases of our approach: constraint elicitation, constraint specification, consistency analysis, and code generation. As a result, we have created a consistent OCL specification without having written a single line of OCL.

7.3 Summary

We have presented COPACABANA, an implementation of our pattern approach to developing consistent constraint specifications. We implemented COPACABANA as an extension to IBM Rational Software Architect (RSA). It provides effective tool support for constraint elicitation, constraint specification using constraint patterns, consistency analysis for pattern-based constraint specifications, and code generation.

Since we built COPACABANA on top of various frameworks in RSA as explained in Section 7.2, the code base for COPACABANA merely contains the application logic and is thus rather small. In total, COPACABANA comprises around 4200 lines of code, which is distributed over 101 classes.

COPACABANA has been published as *IBM Constraint Patterns and Consistency Analysis* on the IBM developerWorks website [IBM, 2007a]. The plug-ins are bundled as a *reusable asset* for RSA; using the Reusable Asset Specification (RAS) explorer in RSA, the plug-ins can be downloaded and used for free. On the website, there is also a tutorial available that guides model developers step-by-step through the installation and the development process.

In the following chapter, we report on larger case studies that we performed using COPACABANA. We elaborate on the details of COPACABANA's implementation in Appendix B.

Chapter 8

Validation

In this chapter, we use the approach developed in this dissertation to develop consistent and concise constraint specifications for real-world models. To this end, we first define a set of criteria that we use to validate our approach in Section 8.1. In the remainder of this chapter, we present three case studies, one in the area of business monitoring (Section 8.2), one in the area of process-model merging (Section 8.3), and one for the "Royal & Loyal" model, which is frequently used as benchmark for MDE approaches (Section 8.4). For each case study, the model and a predominantly informal constraint specification were given initially. Based on the informal specification, we develop a formal specification using constraint patterns and evaluate the results of each case study according to the *quantitative* criteria that we specify in 8.1. In Section 8.5, we summarize the results of the case studies and evaluate our approach according to the *qualitative* criteria listed in the following section.

8.1 Evaluation Criteria

In Chapter 1, we claimed that our approach supports the development of concise and consistent constraint specifications. In the following, we present *quantitative* and *qualitative* evaluation criteria to validate our claim.

Quantitative Criteria.

Specification Coverage. What percentage of a given set of constraints can be expressed using the library of composable constraint patterns from Chapter 4?

Conciseness. To what extent is the pattern-based specification more concise than the same specification in textual languages, e. g., OCL or Java?

Analysis Performance. How does the performance of our analysis method compare to other automatic analysis tools?

Elicitation Coverage. Given a constraint specification, what percentage of it is covered by the constraint-elicitation component? What is the nature of the constraints that are automatically elicited, but do not appear in the constraint specification?

Qualitative Critera.

Analysis Quality. For a pattern-based constraint specification, what is the quality of the consistency analysis regarding false positives and how to deal with them? Are the assumptions in the consistency theorems too weak or too strong?

Limitations. What limitations have we come across in the course of the case studies? Are there disadvantages of our approach compared to traditional approaches?

8.2 Case Study 1: Business Monitoring

In this section, we perform a case study on the monitor model, a meta-model for business monitors used in IBM WebSphere Business *Modeler* (WBM) [IBM, 2007b]. Business monitors monitor events generated from processes, aggregate metrics from these events, and react to predefined situations by issuing events.

The task of the original case study was to implement the constraints from the specification of the monitor model. Most of these constraints were specified informally in English, which caused numerous cases of ambiguity in the specification. A small part of the constraints was formalized in OCL, but the given OCL code was often incomplete or syntactically incorrect.

We worked together with the product development group of WBM on the formalization of the constraints such that instances of the model can be automatically evaluated against the specification. At the end of the day, the team implemented the constraints from the specification in Java, which resulted more than 5,000 lines of complicated code. Furthermore, writing this code involved many repetitive tasks and – because of the complexity of the code – an enormous amount of bug hunting.

This chapter is structured as follows. In Section 8.2.2, we present the meta-model and provide a high-level description of the business monitor's semantics. In Section 8.2.3, we provide a list of constraints from the specification document of the monitor model. Where possible, we provide a formalization of each constraint using the pattern library from Section 4.2. Subsequently, we provide a *quantitative* evaluation of our approach regarding the criteria defined in Section 8.1.

8.2.1 Business Monitoring in a Nutshell.

Business monitors, e. g., IBM WebSphere Business *Monitor* (MON) [IBM, 2007c], are active components that assess the performance of business processes executed on a process server as illustrated in Figure 8.1. They receive events issued by the processes, e. g., the start and termination of tasks, and subsequently process these events. In the course of this processing, business monitors compute values by aggregating data from incoming events and store them in *metrics*. A special type is the key performance indicator (KPI), which is used to store business-critical values.

The behavior of business monitors is determined by monitor models, which define the types of events that the monitors react to, the way that values are aggregated, and the monitors' reaction to predefined situations, e. g., occurrence of critical values. The aggregation of values resembles calculations in spread sheets where values can be written in a certain cell, which itself is the input for another calculation [Frank, 2007].

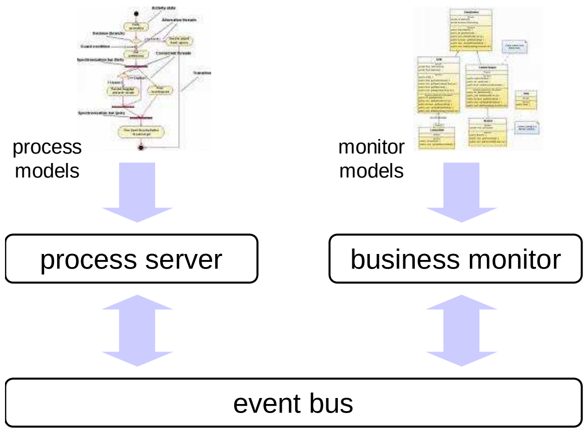


Figure 8.1: Business monitoring in a nutshell.

8.2.2 The Meta-Model for Monitor Models.

As explained, business monitors observe inbound events, calculate metrics from these events, and react to certain situations by issuing outbound events. In total, the meta-model consists of 25 classes and 49 associations. In Figure 8.2, we show the core elements of a monitor model's workflow. Inbound events (InboundEventDefinition) serve as inputs to maps (Map Definition) as a special type of InputslotDefinition . A map is a function that maps a set of inputs to an output (OutputSlotDefinition) according to the function definition in its outputValue. A special kind of output is an output event (OutboundEventDefinition).

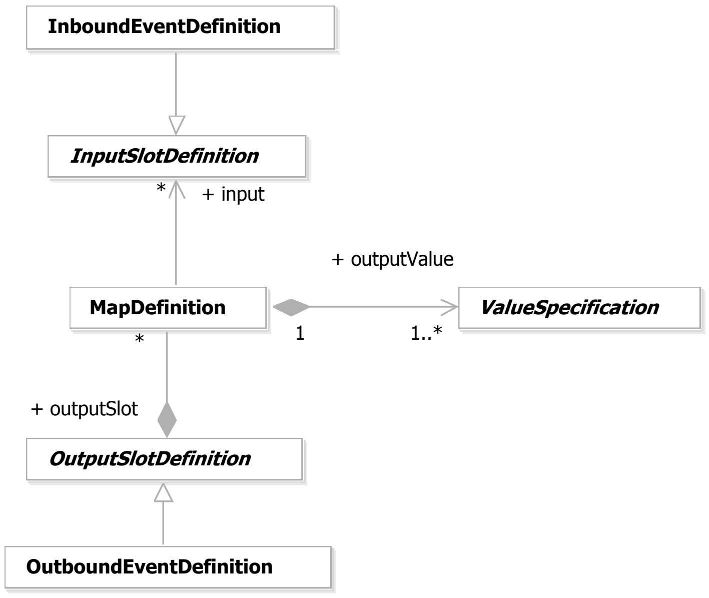


Figure 8.2: Monitor model: maps and their inputs and outputs.

Besides incoming events, there are four more kinds of inputs for maps: metrics (MetricDefinition), keys (KeyDefinition), timers (Timer Definition), and counters (CounterDefinition). The latter three and ReadWriteMetricDefinition can also serve as outputs for maps. Furthermore, there are several special types of metrics: ReadOnlyMetricDefinition, its subclass ExternalMetricDefinition, and KPIDefinition. Figure 8.3 gives an overview of these classes and their relations.

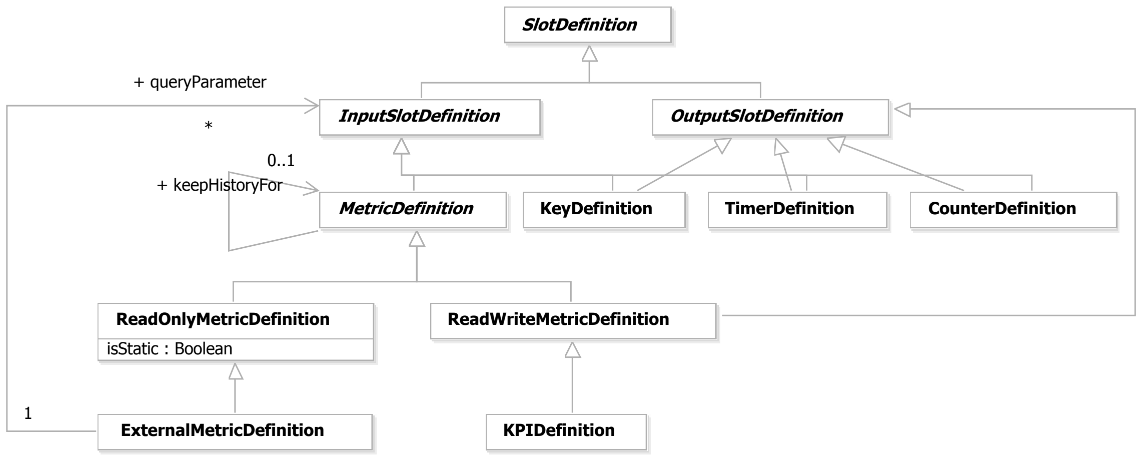


Figure 8.3: Monitor model: metrics.

Besides maps, the behavior of monitors is controlled by situations as shown in Figure 8.4. A situation (SituationDefinition) occurs when a specific event occurs (evaluatedWhen.onEvent), when the value of a certain metric changes (evaluatedWhen.onValueChange), or in specific intervals (evaluatedWhen.ownedEvaluationTime). Upon occurrence of a situation, the computation of a map can be triggered (situationTriggeredMap), an outgoing event (situationTriggeredEvent) can be issued, or a counter can be manipulated.

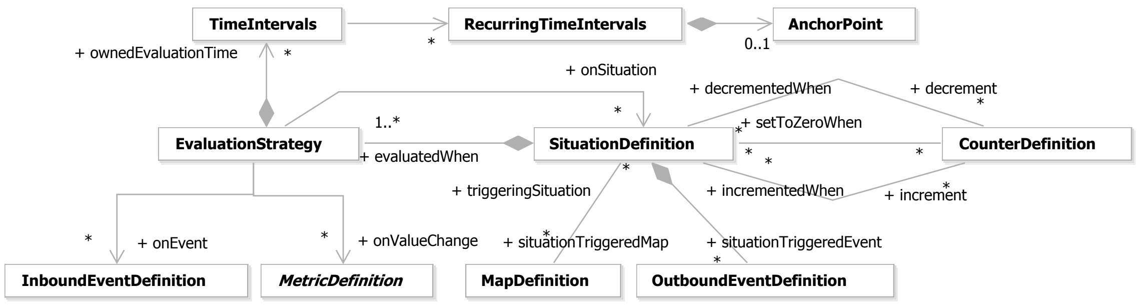


Figure 8.4: Monitor model: situations and triggers.

Metrics, events, and situations can be aggregated in monitoring contexts (MonitoringContextDefinition) as shown in Figure 8.5. A monitoring context is created in the monitor for each instance of the observed business process and can be terminated by certain situations. A hierarchy of monitoring contexts can be created using context relations (ParentContextRelationship). If necessary, business monitors can automatically create parent context definitions if parentContextAutoCreated is set to true.

A MonitoredEntity is a model element representing a permanent real-world thing under observation. A MonitoredEntity references a MonitoringContextDefinition, which is instantiated to reflect the real-world asset that the MonitoredEntity represents when the monitor model is loaded into a monitor, as shown in Figure 8.6. The resulting monitoring context will be the proxy of the real-world asset represented by the MonitoredEntity.

In addition, the monitor model has a type system similar to the UML metamodel [Object Management Group (OMG), 2006c]. Figure 8.7 shows a relevant extract of the model: slots are typed elements, and types can basically be data types or events.

In the monitor model, events can be filtered and correlated using conditions. Fig-

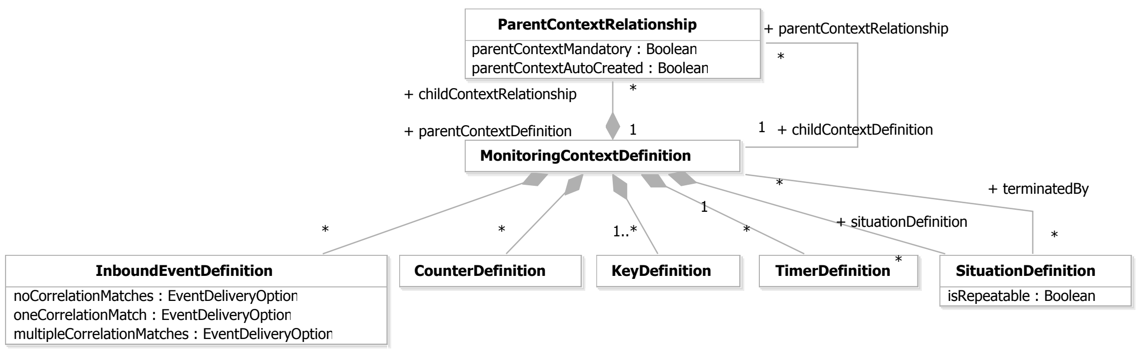

ParentContextRelationship
parentContextMandatory : Boolean
parentContextAutoCreated : Boolean
+ parentContextRelationship
+ childContextRelationship
+ parentContextDefinition
+ childContextDefinition
MonitoringContextDefinition
+ terminatedBy
+ situationDefinition
InboundEventDefinition
noCorrelationMatches : EventDeliveryOption
oneCorrelationMatch : EventDeliveryOption
multipleCorrelationMatches : EventDeliveryOption
CounterDefinition
KeyDefinition
TimerDefinition
SituationDefinition
isRepeatable : Boolean

Figure 8.5: Monitor model: monitoring contexts.

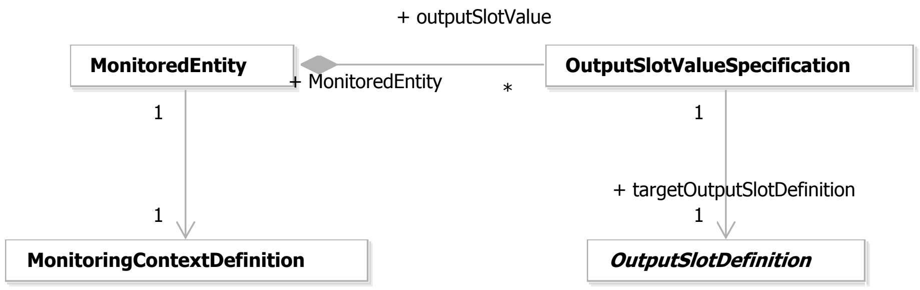

+ outputSlotValue
MonitoredEntity
+ MonitoredEntity
OutputSlotValueSpecification
+ targetOutputSlotDefinition
MonitoringContextDefinition
OutputSlotDefinition

Figure 8.6: Monitor model: monitored entities.

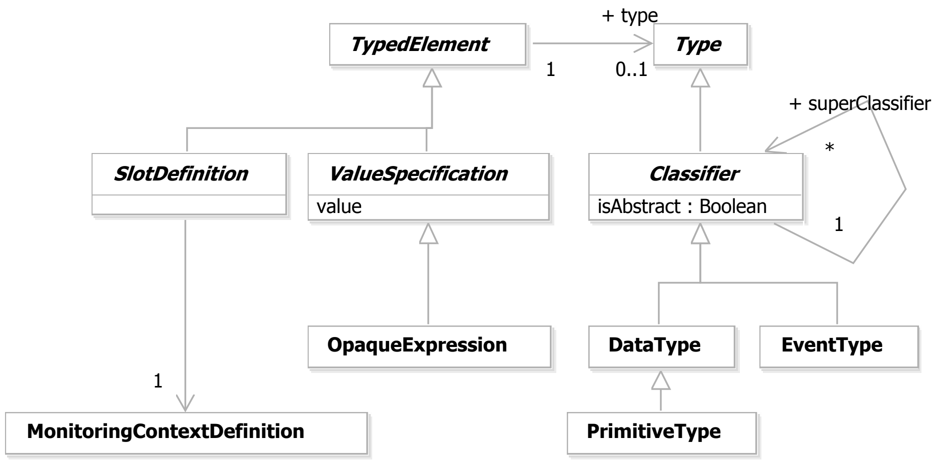


Figure 8.7: Monitor model: types.

ure 8.8 shows the relevant extract of the meta-model.

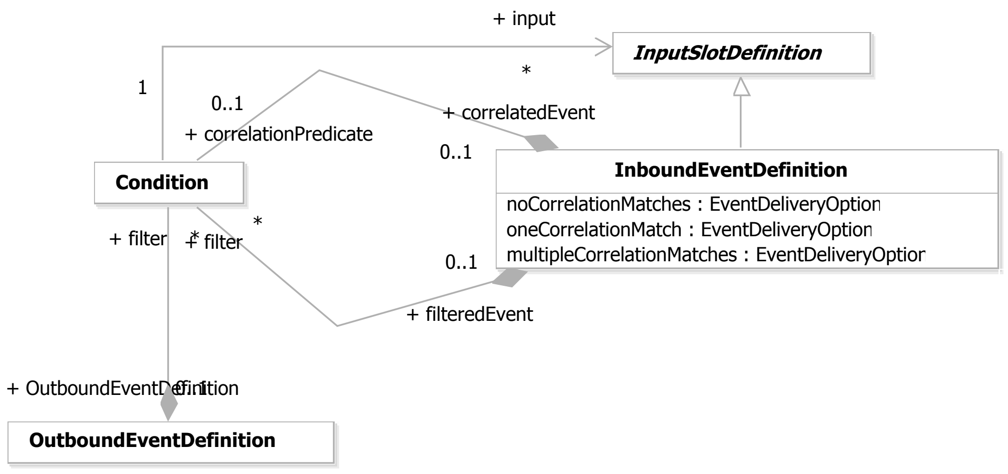


Figure 8.8: Monitor model: filters.

Figure 8.9 shows an example instance of the monitor model. It represents a monitor that counts the number of pages printed on some printer. It reacts to pagePrinted events: upon occurrence of this event, a pagePrintedSituation is triggered because the EvaluationStrategy of the situation is defined through the onEvent association. The counter pageCounter is thus increased on every occurrence of the pagePrinted event. All this elements are grouped in a monitoring context mcd1.

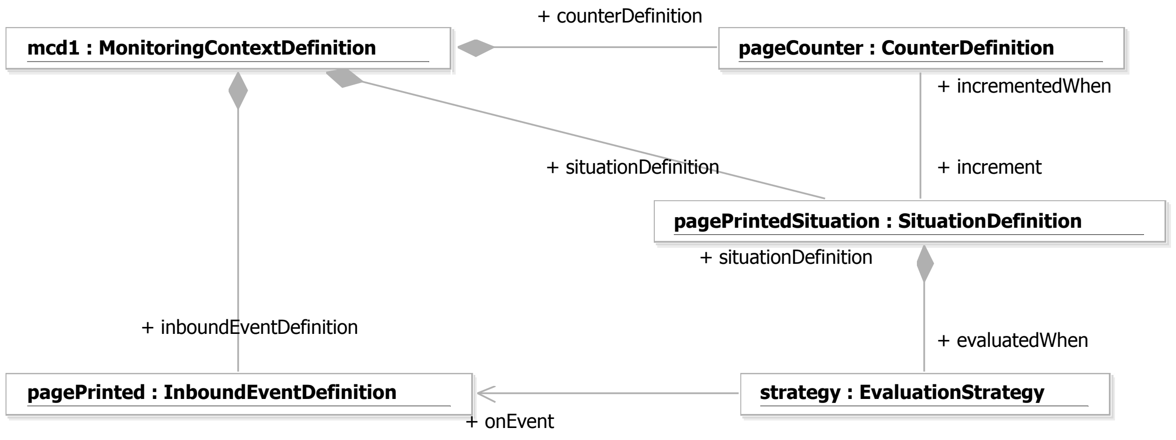

+ counterDefinition
mcd1 : MonitoringContextDefinition
pageCounter : CounterDefinition
+ incrementedWhen
+ situationDefinition
+ increment
pagePrintedSituation : SituationDefinition
+ situationDefinition
+ inboundEventDefinition
+ evaluatedWhen
pagePrinted : InboundEventDefinition
strategy : EvaluationStrategy
+ onEvent

Figure 8.9: Example monitor configuration.

Each monitor model must satisfy certain constraints that ensure that each step in the computation is well-defined and terminates. Furthermore, it needs to be ensured that computations stay within the bound of the correct context definitions. These constraints are listed in a specification document where they are specified informally in English.

In the subsequent section, we show the list of constraints from the specification of the monitor model. For each constraint, we provide the original description in English from the specification document and add a formal definition.

8.2.3 Constraints for the Monitor Model.

For each constraint, we copy its informal description from the specification and formalize it using OCL. Wherever informal constraint descriptions were ambiguous, we consulted the authors of the specification to disambiguate the constraints.

We furthermore specify each constraint using our library of constraint patterns. If the pattern representation is not intuitive, we provide further information that justifies our choice of constraint patterns. Thus, studying this section may help model developers using constraint patterns in their modeling projects.

We structured the constraints according to the central model elements of the monitor model. For making this chapter more readable, we have moved the majority of constraints to Appendix C. This subsection contains some representative constraints and comprises the constraints for maps and events. This subsection is followed by a quantitative evaluation in Section 8.2.4.

8.2.3.1 Maps.

Constraint 8.2.1 (map inputs). *If a map has in inbound event as input, then it must be the only input.*

```
context MapDefinition inv map_inputs:
 self.input->exists(x | x.oclIsTypeOf(InboundEventDefinition))
 implies self.input->size()=1
```

Since this constraint is an implication, we represent it using an instance of the *If-Then-Else* pattern. The assumption is a *Type Relation* and the conclusion is a *Multiplicity Restriction*.

```
context MapDefinition inv map_inputs:
  IfThenElse(TypeRelation(MapDefinition,input,{InboundEventDefinition}),
             MultiplicityRestriction (MapDefinition,input,=,1),
            )
```

Constraint 8.2.2 (map diamonds). *There should be at most one map per metric in the execution path.*

This constraint ensures that monitor models do not include diamond-shaped map configurations, i. e., two maps trigger the computation of a third map, which leads to lack of synchronization.

```
context MapDefinition inv map_diamonds:
  self.input.mapDefinition->forAll(m1, m2 |
                m1.allInputMaps()->intersect(m2.allInputMaps)->isEmpty())
```

This constraint can be specified using the *Unique Path* pattern, which ensures that there is only one path from one map to its predecessors.

```
context MapDefinition inv map_diamonds:
  UniquePath(MapDefinition,input.MapDefinition)
```

Constraint 8.2.3 (gated maps). *If a timer is an input slot of a map then this map must be gated by a situation, i. e., the map must be related to a situation via its* triggeringSituation *association.*

```
context MapDefinition inv gated_maps:
  self.input−>exists( i | i.oclIsTypeOf(TimerDefinition))
                implies self. triggeringSituation −>size() >0
```

The pattern representation of this constraint is analog to *map inputs*.

```
context MapDefinition inv gated_maps:
  IfThenElse(TypeRelation(MapDefinition,input,TimerDefinition),
              MultiplicityRestriction (MapDefinition, triggeringSituation , >, 0),
             )
```

Constraint 8.2.4 (map context). *Each triggeringSituation must be owned by the same MonitoringContextDefinition as the one owning the OutputSlotDefinition of the map.*

```
context MapDefinition inv map_context:
  self. triggeringSituation −>forAll(s |
    s.monitoringContextDefinition = self.outputSlot.monitoringContextDefinition)
```

The pattern representation of this pattern uses the composite pattern *ForAll*.

```
context MapDefinition inv map_context:
  ForAll (MapDefinition, triggeringSituation ,
            AttributeRelation ( SituationDefinition ,monitoringContextDefinition, =,
                         situationTriggeredMap.outputSlot.monitoringContextDefinition))
```

Constraint 8.2.5 (map types). *The expression type of each outputValueSpecification must conform to the type of the targetSlotDefinition.*

```
context MapDefinition inv map_types:
  self.outputValue−>forAll( v | v.type = outputSlot.type)
```

```
context MapDefinition inv map_types:
       AttributeRelation (MapDefinition, outputSlot.type, =,
                          outputValue, type)
```

Constraint 8.2.6 (map multiplicities). *The number of outputValue specifications must be less or equal to the multiplicity of the outputSlot Definition. (Loosely speaking, a MapDefinition must not specify more output values than there are array positions in the target slot.)*

```
context MapDefinition inv map_multiplicities :
  if (not outputSlot.oclIsKindOf(ReadWriteMetricDefinition))
  then outputValue−>size() <= 1
        else let metric = outputSlot.oclAsType(ReadWriteMetricDefinition) in
       metric.mapDefinition−>outputValue−>size()<= metric.upperBound.value
  endif
```

```
context MapDefinition inv map_multiplicities :
  IfThenElse(Negation(TypeRelation(MapDefinition, outputSlot, {ReadWriteMetricDefinition})),
     MultiplicityRestriction (MapDefinition, outputValue, <=, 1),
     MultiplicityRestriction (MapDefinition, outputSlot.MapDefinition, <=,
                             outputSlot.upperBound.value))
```

Constraint 8.2.7 (map parameters). *If the outputSlotDefinition is also a kind of InputSlotDefinition, then it must be amongst the parameters of each outputValue Specification.*

```
context MapDefinition inv map_parameters:
  outputSlot.oclIsKindOf( InputSlotDefinition )  implies
          outputValue−>forAll(input−>includes(outputSlot))
```

For the conclusion of this constraint, we use the *Literal OCL* pattern.

```
context MapDefinition inv map_parameters:
  IfThenElse(TypeRelation(MapDefinition, outputSlot, { InputSlotDefinition }),
                LiteralOcl (MapDefinition, ”outputValue−>forAll(input−>includes(outputSlot))”))
```

Constraint 8.2.8 (map aggregation). *When parent refers to a metric in its child, it can only refer to it using an aggregation function across all the instances of that child, but what if we need to send a specific value from the child to the parent then the map must be owned by the child. A child can own a parent map but a parent cannot own a child map.*

```
context InputSlotDefinition
def: getContext(): MonitoringContextDefinition =
      if  self.oclIsTypeOf(TimerDefinition)
     then self.oclAsType(TimerDefinition).monitoringContext
     else
             if  self.oclIsTypeOf(CounterDefinition)
            then self.oclAsType(CounterDefinition).monitoringContext
      else
                          if  self.oclIsTypeOf(InboundEventDefinition)
             then self.oclAsType(InboundEventDefinition).monitoringContext
       else
               if  self.oclIsTypeOf(MetricDefinition)
              then self.oclAsType(MetricDefinition).monitoringContext
        endif
       endif
      endif
     endif
```

```
context MapDefinition
def: getContext(): MonitoringContextDefinition =
     self.descriptor. classifier −>select(c | c.oclIsKindOf(DescriptorType))−>
     select(d | d.name = 'OwnerMapModelingPropertiesType.MapDefinition').ownedAttribute−>
     select(a | a.name = 'OwningMonitoringContext').slot.value−>select(v |
     v.oclIsKindOf(InstanceValue)).instance. classifier −>any()

inv map_aggregation:
  self.input−>forAll( i | self.getContext() = i.getContext() or
                          self.getContext().getAncestors()−>includes(i.getContext()))
```

Since this constraint employs numerous user-defined functions, we cannot express it using our predefined constraint patterns.

Constraint 8.2.9 (map inputs 2). *Maps only access elements in their input slots.*

```
context MapDefinition inv map_inputs_2:
    map.outputValue−>forAll(value |
       if  (value.oclIsTypeOf(StructuredOpaqueExpression))
       then (value.oclAsType(StructuredOpaqueExpression).getInputs())−>
            forAll ( o | map.input−>includes(o))
       else true
       endif)
```

Since this constraint uses type casts, it cannot be expressed using our constraint patterns. As explained in Section 6.5.1, method calls cannot be used as parameter values for pattern instances.

Constraint 8.2.10 (map outputs). *Maps must not target read-only attributes of output slots.*

```
context MapDefinition inv map_outputs:
  self.outputSlot.type.isPrimitiveType() or
  self.outputValue−>size() <= 1 or
  self.outputValue−>forAll( v | v.oclAsType(InstanceValue).instance.slot−>forAll(s |
     s.definingFeature.isReadOnly and s.value−>size()=0) )
```

```
context Type
def: isPrimitiveType() : Boolean =
     self.oclIsTypeOf(Boolean) or
     self.oclIsTypeOf(Duration) or
     self.oclIsTypeOf(Integer) or
     self.oclIsTypeOf(String) or
     self.oclIsTypeOf(UnlimitedNatural) or
     self.oclIsTypeOf(Real) or
     self.oclIsTypeOf(Time)
```

This constraint employs a user-defined function and can thus not be represented using our patterns.

In Figure 8.10, we show the pattern instances on maps as represented in RSA. The name of the respective constraint can be found in the top compartment of each collaboration. Out of 10 constraints in this section, 7 can be expressed using patterns.

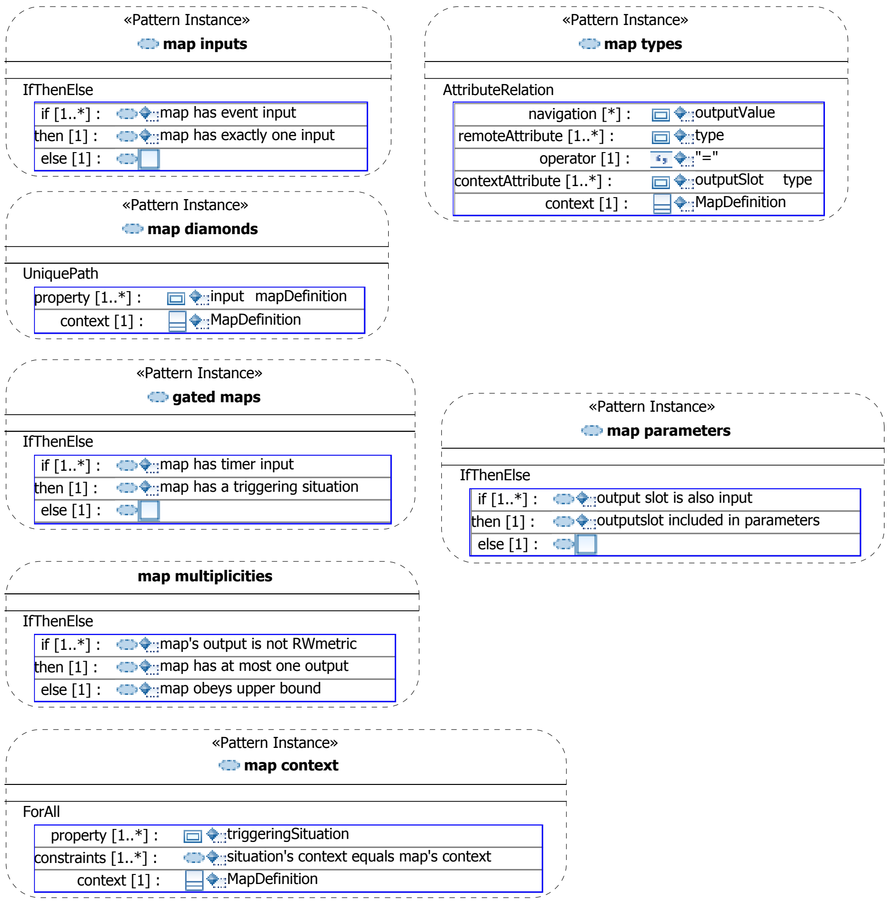

«Pattern Instance»
map inputs
IfThenElse
if [1..*] : :map has event input
then [1] : :map has exactly one input
else [1] :
«Pattern Instance»
map types
AttributeRelation
navigation [*] : :outputValue
remoteAttribute [1..*] : :type
operator [1] : :"="
contextAttribute [1..*] : :outputSlot type
context [1] : :MapDefinition
«Pattern Instance»
map diamonds
UniquePath
property [1..*] : :input mapDefinition
context [1] : :MapDefinition
«Pattern Instance»
gated maps
IfThenElse
if [1..*] : :map has timer input
then [1] : :map has a triggering situation
else [1] :
«Pattern Instance»
map parameters
IfThenElse
if [1..*] : :output slot is also input
then [1] : :outputslot included in parameters
else [1] :
map multiplicities
IfThenElse
if [1..*] : :map's output is not RWmetric
then [1] : :map has at most one output
else [1] : :map obeys upper bound
«Pattern Instance»
map context
ForAll
property [1..*] : :triggeringSituation
constraints [1..*] : :situation's context equals map's context
context [1] : :MapDefinition

Figure 8.10: Pattern instances for maps.

8.2.3.2 Events.

Constraint 8.2.11 (event correlation). *Each inbound event needs to specify the event correlation options.*

```
context InboundEventDefinition inv event_correlation:
  self.noCorrelationMatches−>size() > 0 and
  self.oneCorrelationMatch−>size() > 0 and
  self.multipleCorrelationMatches−>size() > 0
```

We split this constraint into three parts in order to represent it with constraint patterns.

```
context InboundEventDefinition inv event_correlation_1:
   MultiplicityRestriction (InboundEventDefinition,noCorrelationMatches,>,0)

context InboundEventDefinition inv event_correlation_2:
   MultiplicityRestriction (InboundEventDefinition,oneCorrelationMatch,>,0)

context InboundEventDefinition inv event_correlation_3:
   MultiplicityRestriction (InboundEventDefinition,multipleCorrelationMatches,>,0)
```

Constraint 8.2.12 (event type 1). *The type of an inbound event must be a non-abstract EventType.*

```
context InboundEventDefinition inv event_type_1:
  self.type.oclIsKindOf(EventType) and
  (not type.oclAsType(Classifier).isAbstract)
```

```
context InboundEventDefinition inv event_type_1a:
  TypeRestriction(InboundEventDefinition,type,{EventType})

context InboundEventDefinition inv event_type_1b:
  AttributeValueRestriction(InboundEventDefinition,type.isAbstract,=,false)
```

Constraint 8.2.13 (event context). *A triggering situation of each outbound event must be defined in a related context.*

```
context OutboundEventDefinition inv event_context:
  self.monitoringContextDefinition = self.triggeringSituation.monitoringContextDefinition
```

```
context OutboundEventDefinition inv event_context:
  AttributeRelation(OutboundEventDefinition,
                    monitoringContextDefinition,
                    =,
                    triggeringSituation,
                    monitoringContextDefinition)
```

Constraint 8.2.14 (filter type). *The expression of a filter of a correlationPredicate must return a Boolean.*

```
context InboundEventDefinition inv filter_type :
         filter.value->forAll(type=Boolean) and
          correlationPredicate.value->forAll( type = Boolean)
```

```
context InboundEventDefinition inv filter_type_1 :
  ForAll(InboundEventDefinition,
          filter.value,
         {TypeRestriction(ValueSpecification,type,{Boolean})})

context InboundEventDefinition inv filter_type_2 :
  ForAll(InboundEventDefinition,
         correlationPredicate.value,
         {TypeRestriction(ValueSpecification,type,{Boolean})})
```

Constraint 8.2.15 (filter condition). *The filter condition of InboundEventDefinitions can only be parameterized by event content.*

```
context InboundEventDefinition inv filter_condition :
  filter ->forAll(input=Sequence{self})
```

```
context InboundEventDefinition inv filter_condition :
  ForAll(InboundEventDefinition,
          filter ,
         {AttributeValueRestriction(Condition,input,=,Sequence{FilteredEvent})})
```

Constraint 8.2.16 (event type 2). *The type of an InboundEventDefinition must be an EventType (any event received must conform to it).*

```
context InboundEventDefinition inv event_type_2:
  type.oclIsKindOf(EventType)
```

```
context InboundEventDefinition inv event_type_2:
  TypeRestriction(InboundEventDefinition,type,{EventType})
```

Constraint 8.2.17 (correlation value). *Only certain values are permitted for the no / one / multiple correlation-matches attributes of an InboundEventDefinition.*

```
context InboundEventDefinition inv correlation_value:
 Set{'ignore','raiseException','createNewContext'}->includes(noCorrelationMatches) and
 Set{'ignore','raiseException','deliverEvent'}->includes( oneCorrelationMatch ) and
 Set{'ignore','raiseException','deliverToAny',' deliverToAll '}->
   includes(multipleCorrelationMatches)
```

This constraint can be expressed using our patterns, but we need to split it. The reason is that the *Attribute Value Restriction* pattern allows only a single element to be the *value* parameter. We split the constraint twice: First, we turn the explicit conjunction into an implicit conjunction, which results in the constraints correlation_value_1 to correlation_value_3. Second, we use the equivalence $x \in S \equiv \bigvee_i x = s_i$ for each conjunct.

```
context InboundEventDefinition inv correlation_value_1:
Or(AttributeValueRestriction (InboundEventDefinition,noCorrelationMatches,=,ignore),
  AttributeValueRestriction (InboundEventDefinition,noCorrelationMatches,=,raiseException),
  AttributeValueRestriction (InboundEventDefinition,noCorrelationMatches,=,createNewContext))

context InboundEventDefinition inv correlation_value_2:
Or(AttributeValueRestriction (InboundEventDefinition,oneCorrelationMatch,=,ignore),
  AttributeValueRestriction (InboundEventDefinition,oneCorrelationMatch,=,raiseException),
  AttributeValueRestriction (InboundEventDefinition,oneCorrelationMatch,=,deliverEvent))

context InboundEventDefinition inv correlation_value_3:
Or(AttributeValueRestriction (InboundEventDefinition,multipleCorrelationMatches,=,ignore),
   AttributeValueRestriction (InboundEventDefinition,multipleCorrelationMatches,=,
            raiseException),
   AttributeValueRestriction (InboundEventDefinition,multipleCorrelationMatches,=,deliverToAny),
   AttributeValueRestriction (InboundEventDefinition,multipleCorrelationMatches,=,deliverToAll))
```

Similar to Constraint C.4.3, expressing this constraint using our constraint patterns requires significant user sophistication. This motivates the introduction of a new constraint pattern that generalizes the *Attribute Value Restriction* pattern. In Section 8.2.4, we thus introduce the *Attribute Value* Is-One-Of pattern.

Constraint 8.2.18 (map trigger). *An event entry must trigger maps that set all of their contexts' key values.*

```
context InputSlotDefinition
def downStreamSlotDefinitions() : Set(OutputSlotDefinitions) =
   let children = MapDefinition::allInstances()->select(m |
                        m.input->includes(self)).outputSlot   in
   children−>union(children−>select(o | o.oclAsType(InputSlotDefinition).
                                          downStreamSlotDefinition()))

context InboundEventDefinition inv map_trigger:
 noCorrelationMatches = 'createNewContext' implies
        monitoringContextDefinition.keyDefinition−>forAll(
        k | self.downstreamSlotDefinitions()−>includes( k ) )
```

Since this constraint employs numerous user-defined functions, we cannot express it using our predefined constraint patterns. Figure 8.11 shows how the constraints from this section are represented in RSA. Out of 8 constraints in this section, 6 can be expressed using patterns.

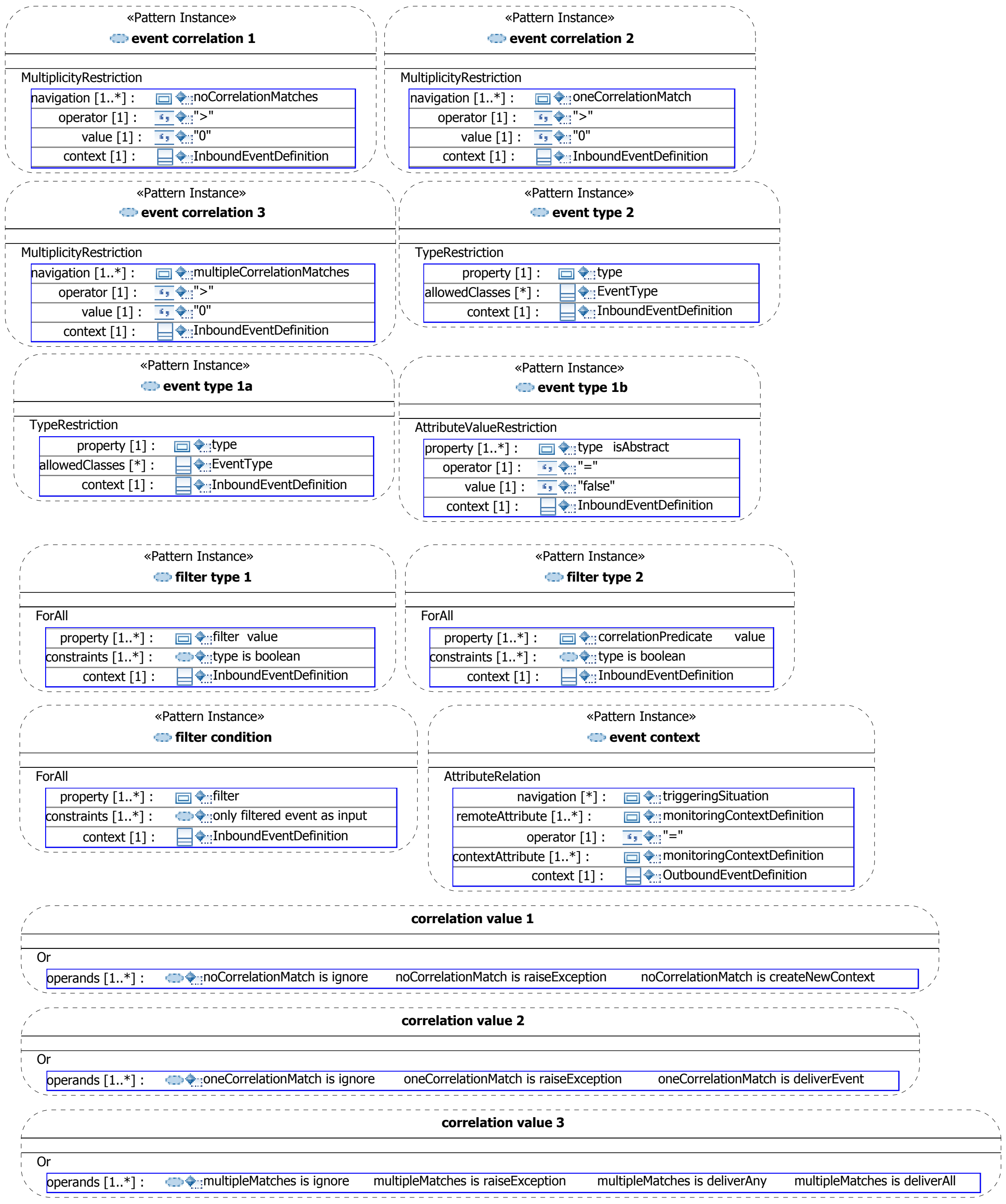

«Pattern Instance»
event correlation 1
MultiplicityRestriction
navigation [1..*] : noCorrelationMatches
operator [1] : ">"
value [1] : "0"
context [1] : InboundEventDefinition
«Pattern Instance»
event correlation 2
MultiplicityRestriction
navigation [1..*] : oneCorrelationMatch
operator [1] : ">"
value [1] : "0"
context [1] : InboundEventDefinition
«Pattern Instance»
event correlation 3
MultiplicityRestriction
navigation [1..*] : multipleCorrelationMatches
operator [1] : ">"
value [1] : "0"
context [1] : InboundEventDefinition
«Pattern Instance»
event type 2
TypeRestriction
property [1] : type
allowedClasses [*] : EventType
context [1] : InboundEventDefinition
«Pattern Instance»
event type 1a
TypeRestriction
property [1] : type
allowedClasses [*] : EventType
context [1] : InboundEventDefinition
«Pattern Instance»
event type 1b
AttributeValueRestriction
property [1..*] : type isAbstract
operator [1] : "="
value [1] : "false"
context [1] : InboundEventDefinition
«Pattern Instance»
filter type 1
ForAll
property [1..*] : filter value
constraints [1..*] : type is boolean
context [1] : InboundEventDefinition
«Pattern Instance»
filter type 2
ForAll
property [1..*] : correlationPredicate value
constraints [1..*] : type is boolean
context [1] : InboundEventDefinition
«Pattern Instance»
filter condition
ForAll
property [1..*] : filter
constraints [1..*] : only filtered event as input
context [1] : InboundEventDefinition
«Pattern Instance»
event context
AttributeRelation
navigation [*] : triggeringSituation
remoteAttribute [1..*] : monitoringContextDefinition
operator [1] : "="
contextAttribute [1..*] : monitoringContextDefinition
context [1] : OutboundEventDefinition
correlation value 1
Or
operands [1..*] : noCorrelationMatch is ignore noCorrelationMatch is raiseException noCorrelationMatch is createNewContext
correlation value 2
Or
operands [1..*] : oneCorrelationMatch is ignore oneCorrelationMatch is raiseException oneCorrelationMatch is deliverEvent
correlation value 3
Or
operands [1..*] : multipleMatches is ignore multipleMatches is raiseException multipleMatches is deliverAny multipleMatches is deliverAll

Figure 8.11: Pattern instances for events.

8.2.4 Quantitative Evaluation.

In this subsection, we perform a quantitative evaluation of the patterns approach for the monitor meta-model. We use the following criteria as defined in Section 8.1: specification coverage, conciseness, performance, and elicitation coverage.

Specification Coverage.

In total, there are 71 constraints, of which 51 constraints can be expressed using the current set of patterns. 20 constraints cannot be expressed using patterns. This means that around 70% of all patterns in this case study can be expressed using patterns, whereas 30% cannot.

To express these 51 constraints, 116 instances of constraint patterns are required, which are used in composite patterns to a large extent. Table 8.1 displays how often individual patterns are instantiated.

Occurrences	Pattern
24	*Multiplicity Restriction*
17	*Attribute Value Restriction*
11	*Attribute Relation*
9	*ForAll*
9	*Literal OCL*
8	*Type Restriction*
8	*If-Then-Else*
7	*Type Relation*
7	*Or*
5	*No Cyclic Dependency*
5	*Unique Identifier*
2	*Object In Collection*
2	*Unique Path*
2	*Negation*

Table 8.1: Pattern instances in the monitor model specification.

Although the majority of constraints can be represented using instances of constraint patterns, some constraints cannot. Some constraints can be represented using a combination of pattern instance and literal OCL expression. For instance, seven instances of the *Literal OCL* pattern that we used contain simple OCL expressions only such as `self.anchorPoint.oclIsUndefined()` or `self.oclIsTypeOf(StructuredOpaqueExpression)`.

In Table 8.2, we present a list of constraints that we cannot express using constraint patterns and briefly explain why this is the case. Of the 20 constraints that we cannot express, two constraints could be expressed by introducing a new pattern to our library of patterns. For Constraints C.10.3 and C.10.4, we introduce the pattern *NoInstances* and define it as follows.

```
pattern NoInstances(class:Class) =
  class::allInstances()->isEmpty()
```

Constraint(s)	Description
8.2.8	We need to define a recursive function to compute the ancestors of a monitoring context. Similarly, the following constraints require the definition of user-defined functions: 8.2.9, 8.2.10, C.2.7, C.2.8, C.2.13, C.2.14, C.3.7, C.5.2, 8.2.18, C.5.1, C.6.1, C.6.2, and C.7.3.
C.2.3	This constraints requires unique values for attributes. However, as explained, it is not covered by the *Unique Identifier* pattern because the constraint requires only a certain subset of all counters to have unique names.
C.2.5	Comprises nontrivial parsing of recursive data structure. This is a domain-specific task and can thus not be generalized in a pattern.
C.2.6,C.8.3	In the *ForAll* pattern, the predicate for the quantified variable has a fixed structure. Thus, not all universally quantified formulas can be expressed using the pattern. Since this constraint requires a navigation from quantified elements to the parent element, it cannot be expressed.
C.10.3,C.10.4	There is no pattern that prevents that a class can be instantiated. We therefore discuss introducing a new pattern, *No Instances*.

Table 8.2: Constraints not expressible using our patterns.

Using this pattern, we can specify Constraints C.10.3 and C.10.4 as follows.

```
context MonitorModel inv no_external_values:
  NoInstances(ExternalMetricDefinition)

context MonitorModel inv no_data_entries:
  NoInstances(DataEntryFieldDefinition)
```

Thus, by adding one new pattern, the percentage of constraints that can be expressed using patterns could be raised from around 70% to around 75% for our case study. However, models to which the *NoInstances* pattern is applied can be neither strongly-consistent nor class-consistent. If a flexible approach is desired, such weak consistency could be allowed in early phases of development, but a stronger notion of consistency should be established when models are finalized. Weak consistency also plays an important role in our second case study in Section 8.3.

New constraint patterns can also be introduced to allow model developers to write more concise specifications. For example, Constraint 8.2.17 could be expressed using existing constraint patterns, but the resulting expression was more complicated than the original constraint. Thus, we introduce the *Attribute Value* Is-One-Of pattern, which we define as follows.

```
pattern AttributeValueIsOneOf(property:Property,set:Set(OclAny)) =
  set−>includes(property)
```

Using this new pattern, Constraint 8.2.17 can be expressed as follows.

```
context InboundEventDefinition inv correlation_value:
  AttributeValueIsOneOf(noCorrelationMatches,
                          Set{'ignore', 'raiseException', 'createNewContext'}) and
  AttributeValueIsOneOf(oneCorrelationMatch,
                          Set{'ignore', 'raiseException', 'deliverEvent'}) and
  AttributeValueIsOneOf(multipleCorrelationMatches,
                          Set{'ignore', 'raiseException', 'deliverToAny', ' deliverToAll '})
```

Conciseness.

In this subsection, we compare the pattern-based specification approach to other approaches in terms of conciseness. To this end, we have implemented the constraint specification for the monitor model in Java, which was a time-consuming and error-prone endeavor. This Java code validates monitor models against the constraints and comprises around 3500 lines of code.

Although many developers have a preference for code, we encourage developing constraint specifications in a concise declarative language instead of Java code for model validation for the following reasons. First, the vast quantity of code necessary to implement the constraints takes significant amount of time to be developed; in addition, maintaining such code is difficult: Upon changes in the model or in the constraints, the code needs to be analyzed for necessary changes. Second, many constraints require navigating from the context object to related objects, as represented by self.x.y in OCL. However, such navigation operations require nested loops in Java which result in verbose and complicated statements. Third, developing model validation code in Java is error-prone due to the extent and the intricacy of the code.

In contrast, constraints written in OCL are less verbose and intricate because OCL has been developed as a concise declarative language for object-oriented models. However, apparently simple facts can result in complicated OCL expressions. For instance, Constraint C.2.1 requires that there can only be one relation between two monitoring contexts, but two existential quantifiers are needed for its OCL formalization. Thus, expertise in formal languages is required to formalize constraints in OCL, and the maintenance of OCL constraints needs both expertise and significant effort.

Constraint specifications developed using our constraint patterns promise to be even more concise. In total, we required 116 pattern instances to express all constraints from the monitor model specification. Most pattern instances can be defined using between two and five lines of code, which results in significantly more concise specifications.

For illustration, we provide three different formalizations of Constraint C.2.1. First, we define it in OCL, second, we provide a Java method that evaluates to true if the constraint holds, and third, we specify the constraint using our patterns.

```
context ParentContextRelationship inv context_relations:
not ParentContextRelationship.allInstances()
      ->exists(x, y | x<>y and x.childContextDefinition=y.childContextDefinition and
                                             x.parentContextDefinition=y.parentContextDefinition)
```

```
public boolean validateContextRelation(List<ParentContextRelationship> allRelations) {
  for (ParentContextRelationship x : allRelations)
    for (ParentContextRelationship y : allRelations) {
      if (x!=y &&
          x.getChildContextDefinition()==y.getChildContextDefinition() &&
          x.getParentContextDefinition()==y.getParentContextDefinition())
        return false; }
    return true; }
```

```
context ParentContextRelationship inv context_relations:
  UniquePath(MonitoringContextDefinition,childContextRelationship.childContextDefinition)
```

Certain types of constraints benefit in particular from a pattern approach, such as constraints on reflexive associations because such constraints are typically complex, but show very similar structure. Other types of constraints are difficult to express using patterns, in particular constraints that are specific to a certain domain or involve user-defined functions. One example is Constraint 8.2.17, a rather simple constraint for which 13 pattern instances are necessary to express it.

Constraints that can be expressed more concisely in OCL than using constraint patterns leave two choices. First, they can be generalized and thus, give rise to new constraint patterns. Second, if they are part of more complex expressions, they can be used in composite constraint patterns using the *Literal OCL* pattern, which wraps arbitrary OCL expressions. However, as discussed in Section 6.5, arbitrary OCL constraints cannot be analyzed by our analysis.

Analysis Performance.

Running consistency analysis on the fully constrained monitor model takes about four seconds on a common personal computer. It checks all 116 pattern instances used, which means that constraints that are composed of multiple constraint patterns can give rise to multiple warnings. This is a conscious design choice and can also be changed in our analysis algorithm such that at most one warning can be raised for each composite constraint.

From the 116 pattern instances, 86 can be shown consistent by our analysis. For the remaining 30 constraints for which the analysis issued warnings, we have performed a secondary analysis (cf. Section 6.6) as follows. For each warning, we created a model state that satisfies the respective set of pattern instances and thus identifies the warning as false positive. This manual analysis showed that all warnings are not actual problems, i. e., the model is consistent. Note that we did not carry out this secondary analysis in a formal way. However, we envision to use to complement our analysis approach with a witness creation approach as discussed in Section 6.1. Besides a higher degree of automation, such an extension can provide a formal and, thus, more reliable analysis result.

For further evaluating our approach to consistency analysis, we performed the consistency analysis with another tool for (almost) automatic consistency analysis of UML/OCL models, USE [Gogolla et al., 2005]. To this end, we have implemented a simple model transformation that transforms models in the RSA representation into the textual format of USE and generates a script for generating model instances.

The first run of USE on the monitor model was rather disillusioning: the tool was not able to validate the pure class model without having added any OCL constraint. In fact, the consistency analysis of USE did not even terminate. After consulting the developers of USE, it is clear that the current version of USE cannot cope with such a large number of associations as in the monitor model. After all, our USE specification contains 76 classes and 84 associations.

The second tool for automatic consistency analysis, UML2Alloy [Bordbar and Anastasakis, 2005], is not suitable for analyzing the company model and its specification because of the limitations of its current version. It does not support multiple inheritance, but for example, TimerDefinition in the meta-model is both a OutputSlotDefinition and an InputSlotDefinition. Further, the return type of operations in UML2Alloy must be either boolean or void, and in addition, operations must not have any parameters. Therefore, most constraints that require user-defined functions cannot be translated by UML2Alloy, e. g., Constraint C.2.13.

Elicitation Coverage.

Although the constraint specification for the monitor was already given and thus, no constraint elicitation had to be performed, we will perform the constraint elicitation on the unconstrained monitor model and once again on the constrained model. Subsequently, we compare the results and draw conclusions.

For the initial unconstrained model, our analysis provided 272 suggestions for refinement. After refining the model with the constraints from the specification, the analysis reported 198 remaining suggestions, which provides two interesting insights.

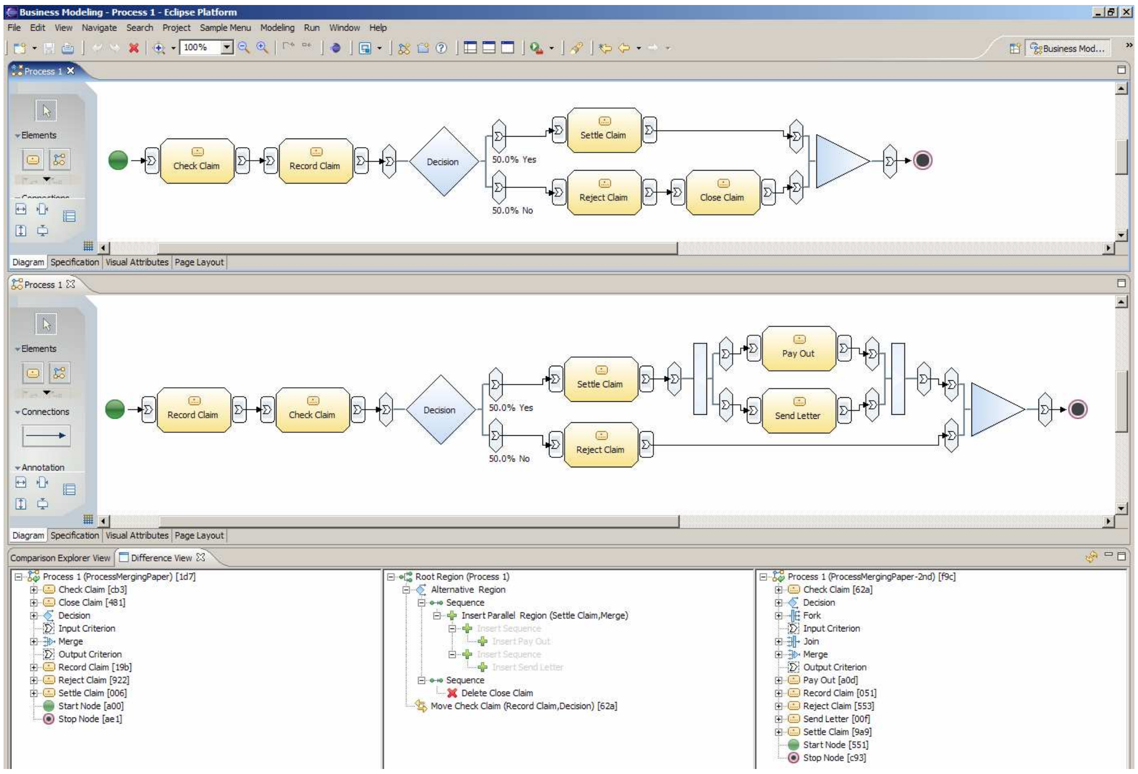
First, the constraint specification covers only around 25% of the possible problems that our analysis finds. Since our analysis provides an over-approximation, i. e., it searches for *potential* problems, it seems natural that only a fraction of these suggestions is actually carried out. Note that the remaining 198 suggestions that are not covered by the specification contain a large number of reflexive relations. We consider reflexive relations one of the most important modeling concept that requires refinement because reflexive relations can cause cycles in the object graph, which in turn can result in nonterminating computations. Although certain cyclic dependencies may not pose a problem in the implementation, we suggest browse the list of elicitation results and extend the specification with instances of the *No Cyclic Dependency* pattern where necessary.

Second, the specification contains constraints not suggested by the analysis. This is caused by the fact that our analysis searches for problems that are independent of the application domain of the model, whereas the constraint specification for the monitor model contains domain-specific constraints. As shown in Chapter 3, domain-specific constraints are elicited manually by domain experts using domain analysis. For example, constraint Constraint 8.2.2 requires that there must be at most one path between to objects of class MapDefinition.

8.3 Case Study 2: Merging Process Models

In this section, we present a case study on constraints for process models that are internally used by IBM WebSphere Business *Modeler* (WBM) [IBM, 2007b]. These constraints originate from a research team in the IBM Zurich Research Laboratory working on process merging, i. e., merging different versions of one process model into a common model, or merging an as-is and a to-be model. The first version of the process-merging prototype did not support all process models, but a large subset. This subset is characterized by a set of restrictions on input models that were originally specified in natural language and as Java code.

In Figure 8.12, we present a screenshot of the process-merging prototype in WBM. The screenshot comprises three main parts: In the upper third and the middle third, two versions of a business process are modeled. In the bottom third of the window, a *difference view* shows the differences between the models and suggests different kinds of actions to the user to unify the two versions.



Process 1 (ProcessMergingPaper) [1d7]
- Check Claim [cb3]
- Close Claim [481]
- Decision
- Input Criterion
- Merge
- Output Criterion
- Record Claim [19b]
- Reject Claim [922]
- Settle Claim [006]
- Start Node [a00]
- Stop Node [ae1]

Root Region (Process 1)
- Alternative Region
  - Sequence
    - Insert Parallel Region (Settle Claim,Merge)
      - Insert Sequence
        - Insert Pay Out
      - Insert Sequence
        - Insert Send Letter
  - Sequence
    - Delete Close Claim
- Move Check Claim (Record Claim,Decision) [62a]

Process 1 (ProcessMergingPaper-2nd) [f9c]
- Check Claim [62a]
- Decision
- Fork
- Input Criterion
- Join
- Merge
- Output Criterion
- Pay Out [a0d]
- Record Claim [051]
- Reject Claim [553]
- Send Letter [00f]
- Settle Claim [9a9]
- Start Node [551]
- Stop Node [c93]

Figure 8.12: Screenshot of the process-merging prototype in WBM.

In this case study, we formally capture the restrictions for this subset using our patterns approach. This section is structured as follows. In Section 8.3.1, we introduce the part of the process meta-model that is relevant for this case study. In Section 8.3.2, we present the limitations on input models for the process-merging prototype in the form of constraints using the patterns approach developed in this thesis. In Section 8.3.3, we provide a *quantitative* evaluation of the results.

8.3.1 The Process Model.

In this section, we introduce the process model, which is used for describing business processes. It is based on concepts used in UML Activity Diagrams [Object Management Group (OMG), 2006c]. We give an overview of the most important concepts and introduce further details in the remainder of this section where needed.

Figure 8.13 and Figure 8.14 show class diagrams of the process model's meta-model, which consists of 33 classes and 43 associations in total. Figure 8.13 gives an overview of different kinds of activity nodes. In general, there are two kinds of activity nodes, ExecutableNode and ControlNode. An executable node can either be an Action, i. e., an atomic task within the business process, or a StructuredActivityNode such as a LoopNode, which can aggregate other activity nodes. A ControlNode can be either an InitialNode or a FinalNode. There are two specialized subclasses of FinalNode: Whereas instances of FlowFinalNode terminate the execution of a single branch, instances of TerminationNode terminate the whole process.

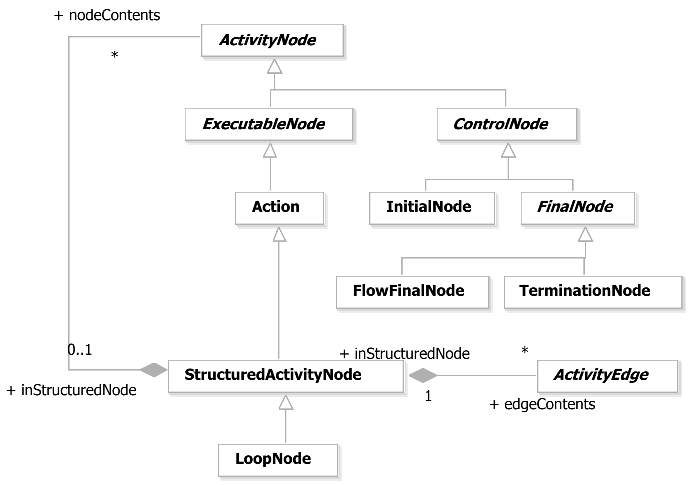


Figure 8.13: Process model: activity nodes.

Nodes are connected by a Pin mechanism. As depicted in Figure 8.14, each action has a set of input and output pins for control flow and a set of input and output pins for object flow. Two pins can be connected by an ActivityEdge; control pins are connected by a ControlFlow and object pins are connected by an ObjectFlow.

Object flow and control flow can be split by using a ControlAction, which is a special type of Action. As depicted in Figure 8.15, there are four types of control actions: Decision and Merge are used for modeling alternative branches in a process, whereas Join and Merge are used for modeling parallel branches.

Figure 8.16 shows an example process model with a single action, *Receive Order*. This action is connected to an initial node by a control flow edge and an input control pin and it is connected to a flow final node by an output control pin and another control flow edge.

8.3.1.1 Semantics in a Nutshell.

Process models are typically executed on process servers. The execution of process models follows a token-flow semantics. Initially, each initial node in the process receives a token and passes it on via its target edge to a ConnectableNode. If this node is a FinalNode, the

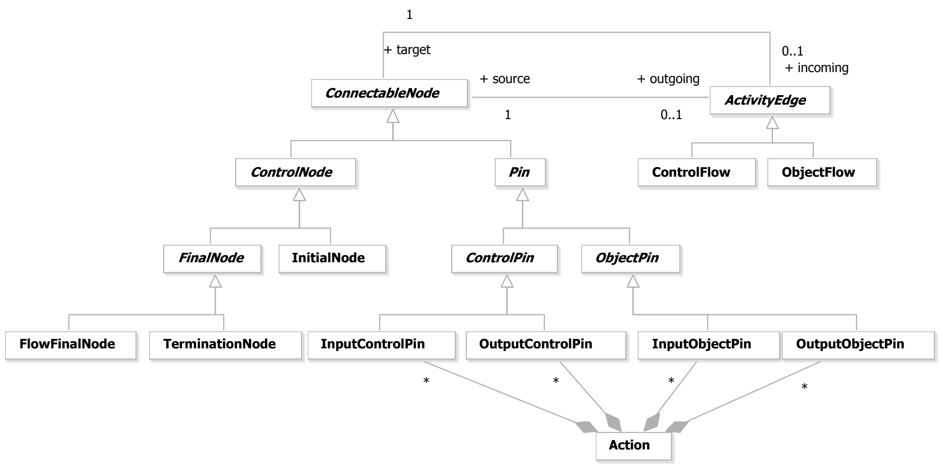


Figure 8.14: Process model: connectable nodes.

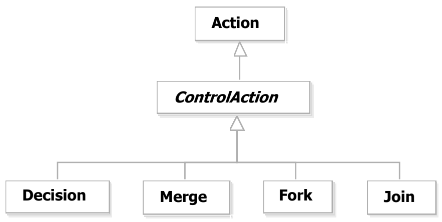


Figure 8.15: Process model: control actions.

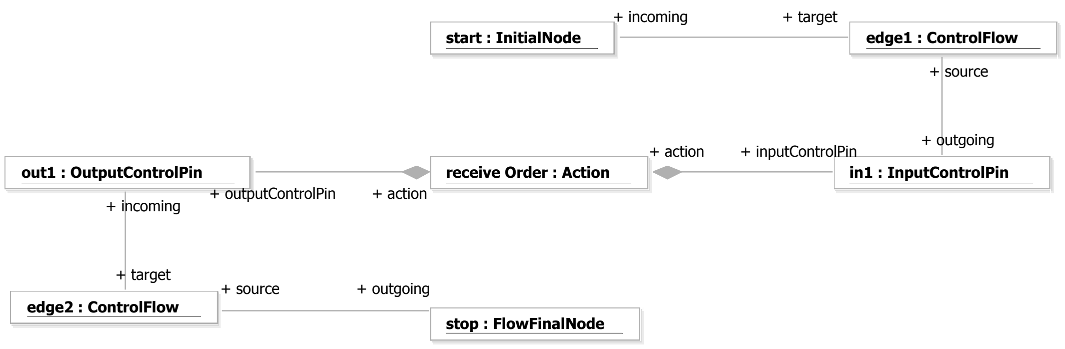


Figure 8.16: Example process model.

token is consumed and if it is a Pin, the action owning the pin is executed, which eventually passes the token on to all outgoing pins of the action. In Figure 8.16, this semantics causes *Receive Order* to be executed exactly once.

There are several exceptions to this simple kind of token flow. On the one hand, tokens can carry objects, which can be passed on using ObjectFlow edges and object pins. Furthermore, token flow can be split using control actions (Figure 8.15). Decision nodes can be used to split control flow into alternative branches, whereas Merge nodes merge previously split alternative branches. Parallel branches can be modeled analogously by using Fork and Join actions. Besides this *explicit* split of the token flow, the token flow can be split and joined *implicitly* by attaching more than one output pin and input pin to an action. This adds an implicit fork and join to the action. Implicit decisions and merges can be modeled similarly by using *pin sets*, which are however not relevant for this case study and are therefore omitted.

8.3.2 Constraints for the Process Model.

In this section, we present the constraints that process models need to satisfy such that they can be correctly processed by the process-merging prototype. We specify each both in OCL and using our library of constraint patterns. For each constraint, if we do not consider the pattern representation to be intuitive, we provide further information that justifies our choice of constraint patterns.

Constraint 8.3.1 (connected models). *Only connected models are supported, i.e., every element is reachable from the start node* and *an end node is reachable from every element.*

```
context StructuredActivityNode inv connected_models:
  self.getPredecessors()->exists(n | n.oclIsTypeOf(InitialNode)) and
  self.getSuccessors()->exists(n | n.oclIsTypeOf(FinalNode))

context Action
def: getPredecessors() : Set(Action) =
  self.inputControlPin.incoming.source.action->
  union(self.inputControlPin.incoming.source.action.getPredecessors())

def: getSuccessors() : Set(Action) =
  self.outputControlPin.outgoing.target.action->
  union(self.outputControlPin.outgoing.target.action.getSuccessors())
```

We can express the constraint *connected models* using the *Type Relation* pattern and involving the *closure* pattern as follows.

```
context StructuredActivityNode
inv connected_models_1:
  TypeRelation(StructuredActivityNode,
               closure(inputControlPin.incoming.source.action),
               {InitialNode})

inv connected_models_2:
  TypeRelation(StructuredActivityNode,
               closure(outputControlPin.outgoing.target.action),
               {FinalNode})
```

Constraint 8.3.2 (no object flow). *Models with object flow are not supported.*

```
context Action inv no_objectflow:
  self.inputObjectPin->isEmpty() and
  self.outputObjectPin->isEmpty()
```

Clearly, this constraint restricts the multiplicity of two properties to zero. Thus, we can express is using two instances of the *Multiplicity Restriction* pattern as follows.

```
context Action
inv no_objectflow_1:
   MultiplicityRestriction (Action,inputObjectPin,=,0)

inv no_objectflow_2:
   MultiplicityRestriction (Action,outputObjectPin,=,0)
```

Constraint 8.3.3 (no termination). *Process models with TerminationNodes are not supported. Use FlowFinalNodes instead.*

```
context ProcessModel inv no_termination:
  TerminationNode::allInstances()->isEmpty()
```

Using the pattern *No Instances* that we defined in the previous case study, we can express this constraint as follows.

```
context ProcessModel inv no_termination:
  NoInstances(TerminationNode)
```

Constraint 8.3.4 (explicit control flow). *Models with implicit forks/joins/decisions/merges are not supported.*

```
context StructuredActivityNode inv  explicit_control_flow :
  self.inputControlPin->size() <= 1 and
  self.outputControlPin->size() <= 1 and
  self.inputObjectPin->size() <= 1 and
  self.outputObjectPin->size() <= 1
```

Each statement in above conjunction can be expressed using the *Multiplicity Restriction* pattern, which results in the following expression.

```
context StructuredActivityNode
inv  explicit_control_flow_1 :
   MultiplicityRestriction (StructuredActivityNode,inputControlPin, <=, 1)

inv  explicit_control_flow_2 :
   MultiplicityRestriction (StructuredActivityNode,outputControlPin, <=, 1)

inv  explicit_control_flow_3 :
   MultiplicityRestriction (StructuredActivityNode,inputObjectPin, <=, 1)

inv  explicit_control_flow_4 :
   MultiplicityRestriction (StructuredActivityNode,outputObjectPin, <=, 1)
```

Constraint 8.3.5 (no loops). *Loop nodes are not processed properly and cannot be merged. Therefore, loop nodes are not supported.*

```
context ProcessModel inv no_loops:
  LoopNode::allInstances()−>isEmpty()
```

Again, we can express this constraint using the *No Instances* pattern as follows.

```
context ProcessModel inv no_loops:
  NoInstances(LoopNode)
```

8.3.3 Quantitative Evaluation.

In this subsection, we perform a quantitative evaluation of the patterns approach for the process meta-model. We use the following criteria as defined in Section 8.1: specification coverage, conciseness, performance, and elicitation coverage.

Specification Coverage.

100% of the constraints for the model-merging prototype can be expressed using our library of constraint patterns. Such a high coverage is caused by the nature of the process-model constraints, which restrict structural properties of the model state and do not restrict attribute values.

It requires 10 pattern instances to specify the five constraints because several constraints comprise several conjuncts, which we each model as distinct pattern instance. In Table 8.3, we summarize which patterns we have used and how often they occur.

Occurrences	Pattern
6	*Multiplicity Restriction*
2	*Type Relation*
2	*No Instances*

Table 8.3: Pattern instances used for the process-merging constraints.

Conciseness.

At first glance, using constraint patterns for expressing the five constraints does not improve conciseness because the lines of code required are comparable in the patterns approach and in plain OCL. However, some improvements are hidden in the details of several constraints.

For example, Constraint 8.3.1 requires the model developer to define two operations that compute transitive closures. By using constraint patterns, he can re-use the *closure* pattern that we initially defined for the *No Cyclic Dependency* pattern. Thus, the model developer substantially saves time and decreases the effort required when the model changes and the constraints need to be adapted.

Constraint 8.3.3 requires knowledge about two OCL operators, of which the first, allInstances(), requires specific knowledge of UML/OCL. In contrast, this constraint can be defined by a single instance of the *No Instances* pattern and setting one parameter value.

Analysis Performance.

Invoking consistency analysis on the constraints for process merging runs for about two seconds. It results in six warnings. Two warnings stem from the two constraints that use the *No Instances* pattern because each instance of this pattern makes a model inconsistent by definition (cf. Section 5.1.1).

The four other warnings are caused by two constraints, *no object flow* and *explicit control flow*, which each constrain the properties inputObjectPin and outputObjectPin. These constraints are not inconsistent, but partly redundant because two conjuncts of *explicit control flow* are implied by *no object flow*. Thus, we can remove them and simplify *explicit control flow* as follows.

```
context StructuredActivityNode inv  explicit_control_flow :
  self.inputControlPin−>size() <= 1 and
  self.outputControlPin−>size() <= 1
```

Elicitation Coverage.

Unlike the case study in Section 8.2, the subject of this case study is not a large set of constraints from a specification document, but a small set of constraints for a very specific problem. Thus, constraint elicitation does not play an important role in this case study.

Nevertheless, we want to briefly present the results of the constraint elicitation component. For the full, unconstrained process meta-model, constraint elicitation issues 405 warnings, which is around 50% more warnings than for the previous case study on the monitor model. The number of warnings does not directly correlate with the number of classes and associations in the models because the process model only has around 33% more classes than the monitor model. In Table 8.4, we summarize the constraint elicitation statistics for both models.

Model	Classes	Associations	Warnings
Process Model	33	43	405
Monitor Model	25	49	272

Table 8.4: Dependency of model size and constraint elicitation warnings.

After adding the five constraints from the process-merging prototype, 7 warnings are eliminated. Thus, some of the constraints can be considered useful for the process model beyond the scope of the prototype. Whereas the constraints that use the *No Instances* pattern are clearly not useful because they render the model inconsistent, we consider constraint Constraint 8.3.1, which requires that every element in a process is reachable from a start and an end node, a candidate for being added to the general constraint specification for process models.

8.4 Case Study 3: Royal & Loyal

In this section, we present a case study on the "Royal & Loyal" model. The Royal & Loyal model is used in [Kleppe and Warmer, 2003] for introducing OCL by example. Since then, it has been used in various publications, e. g., [Tedjasukmana, 2006, Dzidek et al., 2005], and it is shipped with several tools as an example model, e. g., [Dresden Technical University, 2007]. Various constraints of different types are defined for the Royal & Loyal model, which makes it a good benchmark for MDE approaches such as ours.

This section is structured as follows. In Section 8.4.1, we present the Royal & Loyal model as introduced in [Kleppe and Warmer, 2003] and formalize its constraints in Section 8.4.2 using our pattern approach. In Section 8.4.3, we provide a *quantitative* evaluation of the results.

8.4.1 The Royal & Loyal Model.

Figure 8.17 illustrates the Royal & Loyal model, which represents a company that handles loyalty programs, i. e., various kinds of bonuses, for third party companies. Its central class is LoyaltyProgram, which connects objects of class Customer to objects of class ProgramPartner. The membership of customers in loyalty programs is modeled using a UML association class Membership. Association classes specialize both associations and classes and thus combine their properties.

Customers have a CustomerCard for each membership in a loyalty program. Using this card, customers can perform Transactions. There are two types of transactions, Burning, which reduces the number of points in the customer's account, and Earning, which increases the number of points.

Program partners offer Services at predefined ServiceLevels. Each membership is associated with exactly one service level. The model further comprises two enumerations, Date and Color, which are used as types for several attributes in the model.

8.4.2 Constraints for the Royal & Loyal Model.

In this subsection, we present the constraints specified in [Kleppe and Warmer, 2003] for the Royal & Loyal model. The constraints are originally specified both in English and in OCL. We use the approach developed in this thesis to further express the constraints using pattern instances.

Customer.

Constraint 8.4.1 (legal age). *Every customer who enters a loyalty program must be of legal age.*

```
context Customer
inv legalAge: age >= 18
```

```
context Customer
inv legalAge: AttributeValueRestriction (Customer,age,>=,18)
```

Constraint 8.4.2 (sizes agree). *The number of valid cards for every customer must be equal to the number of programs in which the customer participates.*

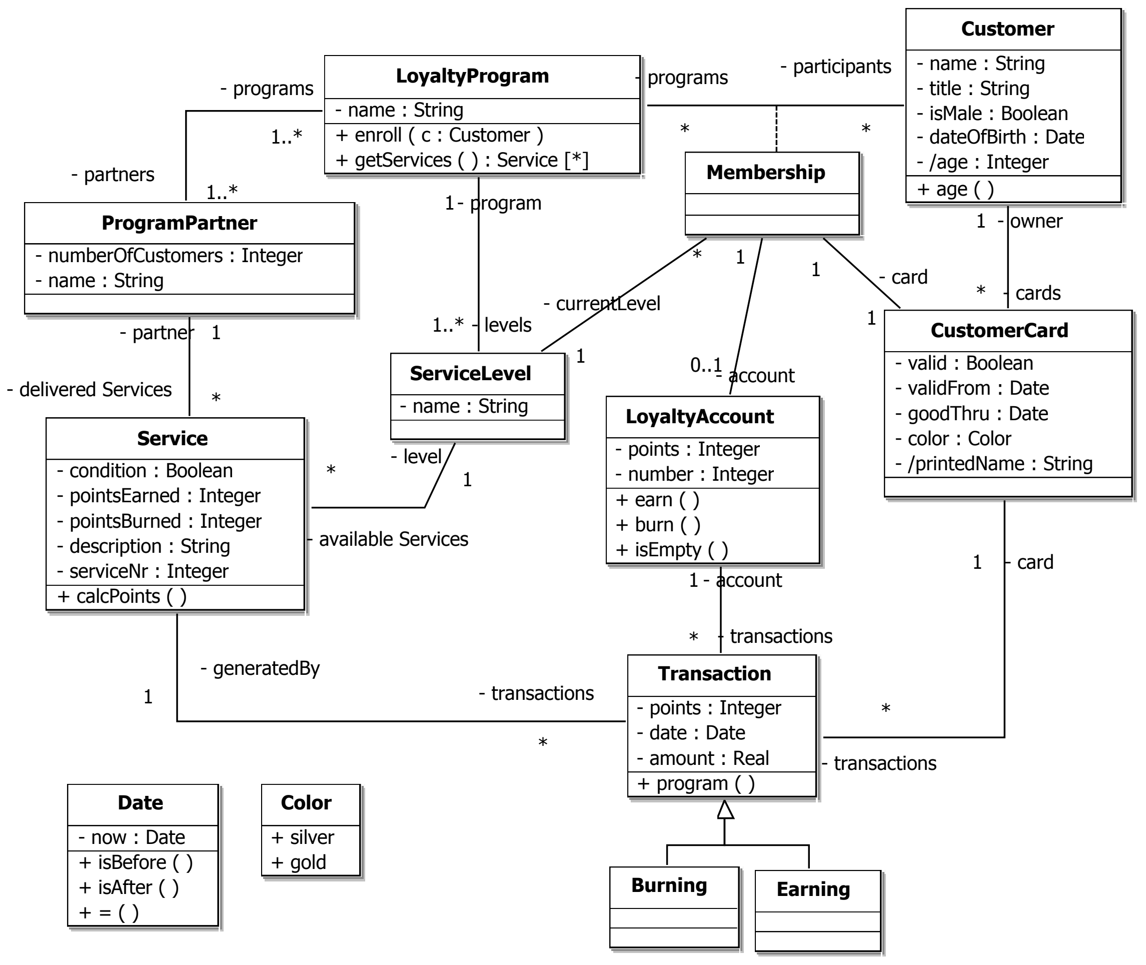


Figure 8.17: The Royal & Loyal model.

```
context Customer
inv sizesAgree:
  programs−>size() = cards−>select(valid=true)−>size()
```

This constraint cannot be expressed using our patterns because there is no pattern in our library for comparing the cardinality of sets.

Constraint 8.4.3 (male title). *Male customers must be approached using the title 'Mr.'.*

```
context Customer
inv maleTitle: isMale implies  title  = 'Mr.'
```

```
context Customer
inv maleTitle: IfThenElse({AttributeValueRestriction(Customer,isMale,=,true)},
                          AttributeValueRestriction(Customer,title,=,'Mr.'),
                         )
```

We show the pattern instances for customers, *legal age* and *male title* in Figure 8.18. Note that *male title* is a composite constraint composed of the constraints *customer is male* and *title is Mr*, indicated by the rectangle.

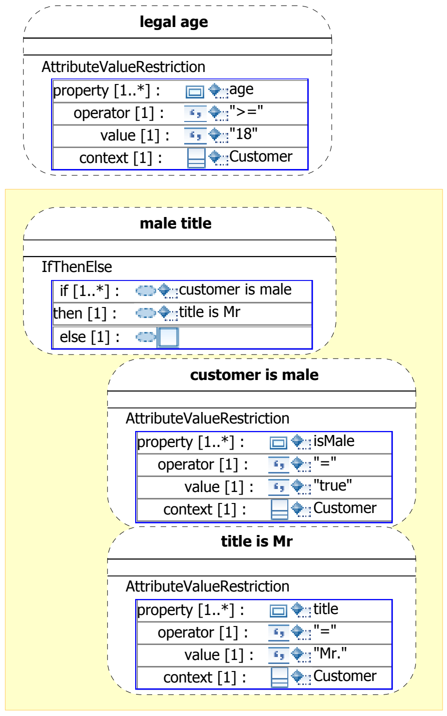


Figure 8.18: Pattern instances for customers.

CustomerCard.

Constraint 8.4.4 (check dates). *The* validFrom *date of customer cards should be earlier than* goodThru.

```
context CustomerCard
inv checkDates: validFrom.isBefore(goodThru)
```

This constraint cannot be expressed using our constraint patterns because there is no pattern that allows developers to use parametrized function calls.

Constraint 8.4.5 (birth date). *The birth date of the owner of a customer card must not be in the future.*

```
context CustomerCard
inv birthDate: self.owner.dateOfBirth.isBefore(Date::now)
```

This constraint cannot be expressed using our constraint patterns because there is no pattern that allows developers to use parametrized function calls.

Constraint 8.4.6 (program participation). *The owner of a customer card must participate in at least one loyalty program.*

```
context CustomerCard
inv programParticipation: self.owner.programs->size() > 0
```

```
context CustomerCard
inv programParticipation:  MultiplicityRestriction (CustomerCard,owner.programs,>,0)
```

Constraint 8.4.7 (transaction points). *There must be at least one transaction for a customer card with at least 100 points.*

```
context CustomerCard
inv transactionPoints: self.transactions−>select(points>100)−>notEmpty()
```

```
context CustomerCard
inv transactionPoints: Exists(transactions,
                              {AttributeValueRestriction(CustomerCard,points,>,100)})
```

In Figure 8.19, we summarize the pattern instances for customer cards.

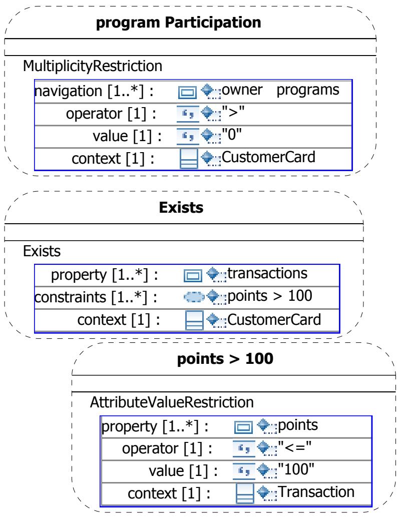

program Participation
MultiplicityRestriction
navigation [1..*] : owner programs
operator [1] : ">"
value [1] : "0"
context [1] : CustomerCard
Exists
Exists
property [1..*] : transactions
constraints [1..*] : points > 100
context [1] : CustomerCard
points > 100
AttributeValueRestriction
property [1..*] : points
operator [1] : "<="
value [1] : "100"
context [1] : Transaction

Figure 8.19: Pattern instances for customer cards.

Membership.

Constraint 8.4.8 (known service level). *The service level of each membership must be a service level known to the loyalty program.*

```
context Membership
inv knownServiceLevel: programs.levels−>includes(currentLevel)
```

```
context Membership
inv knownServiceLevel: ObjectInCollection(Membership,
                                          programs.levels,
                                          currentLevel)
```

Constraint 8.4.9 (correct card). *The participants of a membership must have the correct card belonging to this membership.*

```
context Membership
inv correctCard: participants.cards−>includes(self.card)
```

```
context Membership
inv correctCard: ObjectInCollection(Membership,
                                      participants.cards,
                                      card)
```

Constraint 8.4.10 (level and color). *The color of a membership's card must match the service level of the membership.*

```
context Membership
inv levelAndColor:
  currentLevel.name = 'Silver' implies card.color = Color::silver
  and
  currentLevel.name = 'Gold' implies card.color = Color::gold
```

As before, we split the conjunction into two separate invariants.

```
context Membership
inv levelAndColor1:
  IfThenElse({AttributeValueRestriction(Membership,currentLevel.name,=,'Silver')},
             AttributeValueRestriction(Membership,card.color,=,Color::silver),
            )

inv levelAndColor2:
  IfThenElse({AttributeValueRestriction(Membership,currentLevel.name,=,'Gold')},
             AttributeValueRestriction(Membership,card.color,=,Color::gold),
            )
```

Constraint 8.4.11 (no account). *Memberships must not have associated accounts.*

```
context Membership
inv noAccount: account->isEmpty()
```

```
context Membership
inv noAccount: MultiplicityRestriction(Membership,account,=,0)
```

In Figure 8.20, we summarize the pattern instances for memberships.

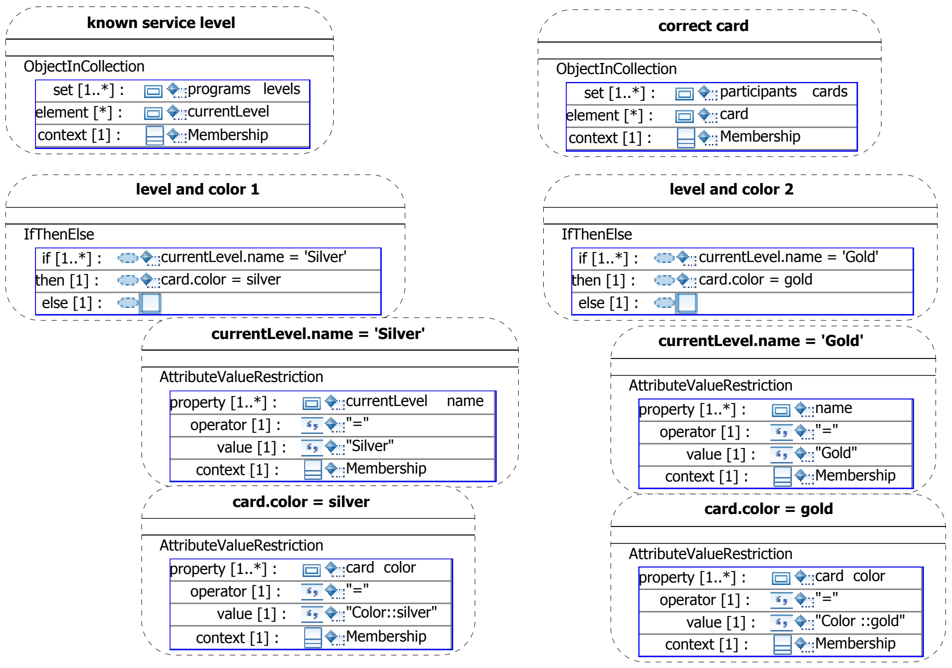

known service level
ObjectInCollection
set [1..*] : programs levels
element [*] : currentLevel
context [1] : Membership
correct card
ObjectInCollection
set [1..*] : participants cards
element [*] : card
context [1] : Membership
level and color 1
IfThenElse
if [1..*] : currentLevel.name = 'Silver'
then [1] : card.color = silver
else [1] :
level and color 2
IfThenElse
if [1..*] : currentLevel.name = 'Gold'
then [1] : card.color = gold
else [1] :
currentLevel.name = 'Silver'
AttributeValueRestriction
property [1..*] : currentLevel name
operator [1] : "="
value [1] : "Silver"
context [1] : Membership
currentLevel.name = 'Gold'
AttributeValueRestriction
property [1..*] : name
operator [1] : "="
value [1] : "Gold"
context [1] : Membership
card.color = silver
AttributeValueRestriction
property [1..*] : card color
operator [1] : "="
value [1] : "Color::silver"
context [1] : Membership
card.color = gold
AttributeValueRestriction
property [1..*] : card color
operator [1] : "="
value [1] : "Color ::gold"
context [1] : Membership

Figure 8.20: Pattern instances for memberships.

LoyaltyProgram.

Constraint 8.4.12 (min services). *Loyalty programs must offer at least one service to their customers.*

```
context LoyaltyProgram
inv minServices: partners.deliveredServices−>size() >= 1
```

```
context LoyaltyProgram
inv minServices:  MultiplicityRestriction  (LoyaltyProgram,partners.deliveredServices,>=,1)
```

Constraint 8.4.13 (no accounts). *If none of the services offered in a loyalty program credits or debits the loyalty accounts, then these instances are useless and should not be present.*

```
context LoyaltyProgram
inv noAccounts: partners.deliveredServices−>forAll(
                    pointsEarned = 0 and pointsBurned = 0 )
                    implies  Membership.account−>isEmpty()
```

This constraint cannot be expressed using our patterns because they do not yet support navigation via association classes as required in this constraint by Membership.

Constraint 8.4.14 (first level). *The name of the first level must be* Silver.

```
context LoyaltyProgram
inv  firstLevel :  levels−>first (). name = ’Silver’
```

This constraint cannot be expressed using our patterns because there is no pattern for assertions on ordered sets.

Constraint 8.4.15 (basic level). *There must exist at least one service level with the name* basic*.*

```
context LoyaltyProgram
inv basicLevel: self. levels−>exists(name = ’basic’)
```

```
context LoyaltyProgram
inv basicLevel: Exists( levels ,{ AttributeValueRestriction (LoyaltyProgram,name,=,’basic’)})
```

Constraint 8.4.16 (max participants). *The number of participants in a loyalty program must be less than 10,000.*

```
context LoyaltyProgram
inv maxParticipants: self. participants −>size() < 10,000
```

```
context LoyaltyProgram
inv maxParticipants:  MultiplicityRestriction  (LoyaltyProgram,participants,<,10000)
```

Constraint 8.4.17 (unique account). *The number of the loyalty account must be unique within a loyalty program.*

```
context LoyaltyProgram
inv uniqueAccount: self.Membership.account−>isUnique(acc | acc.number)
```

This constraint cannot be expressed using our patterns because there is no pattern for expressing that an attribute must be unique for a subset of all objects of a class. In contrast, *Unique Identifier* requires uniqueness for *all* objects of the given class. Thus, the *Unique Identifier* pattern could be used to express a *stronger* constraint.

Constraint 8.4.18 (unique names). *The names of all customers of a loyalty program must be different.*

```
context LoyaltyProgram
inv uniqueNames: self.participants−>forAll(c1,c2 | c1<>c2 implies c1.name <> c2.name)
```

As for Constraint 8.4.17, this constraint cannot be expressed using our patterns because there is no pattern for expressing that an attribute must be unique for a subset of all objects of a class.

Constraint 8.4.19 (max age). *The maximum age of participants in loyalty programs is 70.*

```
context LoyaltyProgram
inv maxAge: participants−>forAll(age() <= 70)
```

```
context LoyaltyProgram
inv maxAge: ForAll(participants,{ AttributeValueRestriction (LoyaltyProgram,age,<=,70)})
```

Constraint 8.4.20 (one account). *There may be only one loyalty account that has a number lower than 10,000.*

```
context LoyaltyProgram
inv oneAccount: self.Membership.account−>one(number < 10,000)
```

This constraint cannot be expressed using our patterns because there does not exist a matching pattern that can be used to express the one() operator.

In Figure 8.21, we summarize the pattern instances for loyalty programs.

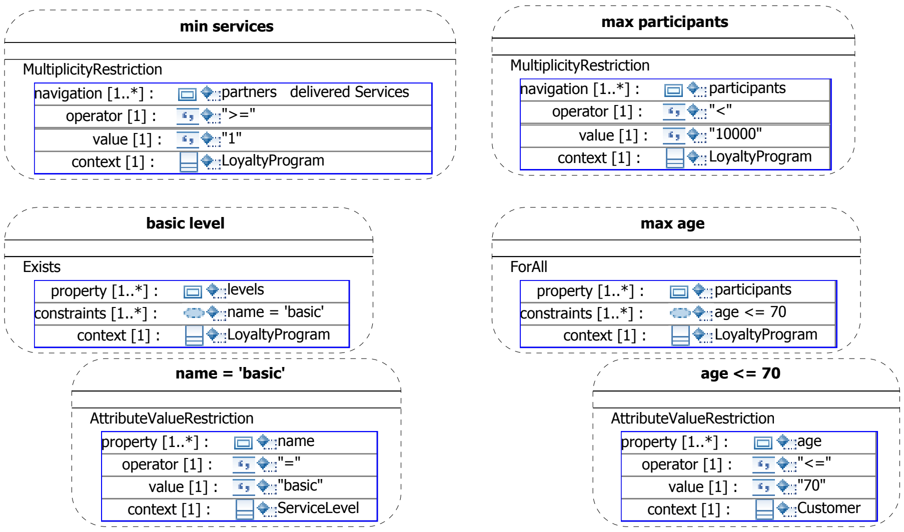

min services
MultiplicityRestriction
navigation [1..*] : partners delivered Services
operator [1] : ">="
value [1] : "1"
context [1] : LoyaltyProgram
max participants
MultiplicityRestriction
navigation [1..*] : participants
operator [1] : "<"
value [1] : "10000"
context [1] : LoyaltyProgram
basic level
Exists
property [1..*] : levels
constraints [1..*] : name = 'basic'
context [1] : LoyaltyProgram
max age
ForAll
property [1..*] : participants
constraints [1..*] : age <= 70
context [1] : LoyaltyProgram
name = 'basic'
AttributeValueRestriction
property [1..*] : name
operator [1] : "="
value [1] : "basic"
context [1] : ServiceLevel
age <= 70
AttributeValueRestriction
property [1..*] : age
operator [1] : "<="
value [1] : "70"
context [1] : Customer

Figure 8.21: Pattern instances for loyalty programs.

ProgramPartner.

Constraint 8.4.21 (nr of participants). *The attribute numberOfCustomers of class ProgramPartner must be equal to the number of customers who participate in one or more loyalty programs offered by this program partner.*

```
context ProgramPartner
inv nrOfParticipants:
  numberOfCustomers = programs.participants−>asSet()−>size()
```

```
context ProgramPartner
inv nrOfParticipants:
   MultiplicityRestriction  (ProgramPartner,programs.participants,=,numberOfCustomers)
```

Constraint 8.4.22 (total points). *A maximum of 10,000 points may be earned using services of one partner.*

```
context ProgramPartner
inv  totalPoints :
    deliveredServices.transactions−>select(oclIsTypeOf(Earning)).points−>sum() < 10,000
```

There is no pattern in our library to express this constraint because of the subexpression select(oclIsTypeOf(Earning)). This expression filters the set of objects accessed through deliveredServices.transactions and there is no pattern that corresponds to such filtering expressions.

In Figure 8.22, we show the only constraint for program partners expressible by our patterns, *nr of participants*.

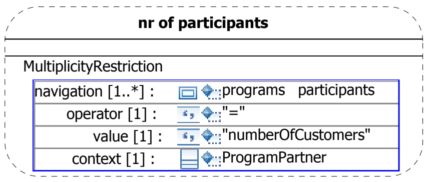


Figure 8.22: Pattern instances for program partners.

LoyaltyAccount.

Constraint 8.4.23 (one owner). *All cards that generate transactions on the loyalty account must have the same owner.*

```
context LoyaltyAccount
inv oneOwner: transactions.card.owner−>asSet()−>size() = 1
```

```
context LoyaltyAccount
inv oneOwner: MultiplicityRestriction (LoyaltyAccount,transactions.card.owner,=,1)
```

Constraint 8.4.24 (positive points). *If the points earned in a loyalty account is greater than zero, there exists a transaction with more than zero points.*

```
context LoyaltyAccount
inv positivePoints : points > 0 implies transactions−>exists(t | t.points > 0)
```

```
context LoyaltyAccount
inv positivePoints :
  IfThenElse({AttributeValueRestriction(points,>,0)},
                Exists(transactions,{ AttributeValueRestriction (LoyaltyAccount,points,>,0)}),
            )
```

Constraint 8.4.25 (500 points). *There must be one transaction with exactly 500 points.*

```
context LoyaltyAccount
inv 500points: transaction.points−>exists(p : Integer | p = 500)
```

```
context LoyaltyAccount
inv 500points: Exists(LoyaltyAccount,transaction.points,{ LiteralOcl (−,"self = 500")})
```

ServiceLevel.

Constraint 8.4.26 (service partner). *The available services for a service level must be offered by a partner of the loyalty program to which the service level belongs.*

```
context ServiceLevel
inv servicePartner: program.partners−>includesAll(self.availableServices.partner)
```

Since there is no constraint pattern in our pattern library that comprises the includesAll operation, we change the context class of this constraint and express it as follows.

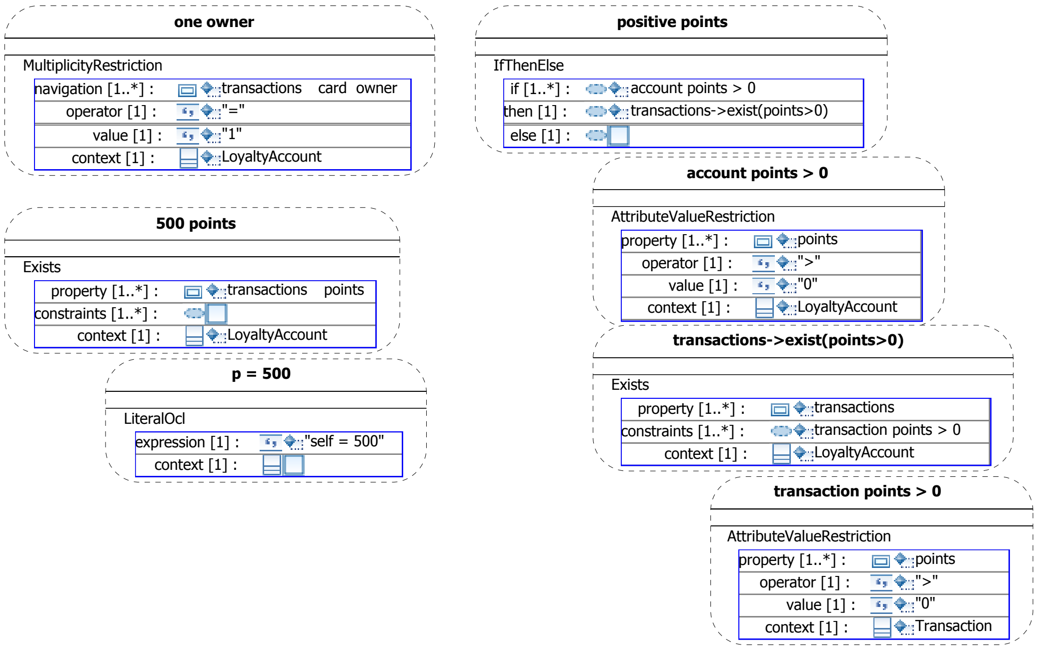

one owner
MultiplicityRestriction
navigation [1..*] : transactions card owner
operator [1] : "="
value [1] : "1"
context [1] : LoyaltyAccount
positive points
IfThenElse
if [1..*] : account points > 0
then [1] : transactions->exist(points>0)
else [1] :
account points > 0
AttributeValueRestriction
property [1..*] : points
operator [1] : ">"
value [1] : "0"
context [1] : LoyaltyAccount
500 points
Exists
property [1..*] : transactions points
constraints [1..*] :
context [1] : LoyaltyAccount
p = 500
LiteralOcl
expression [1] : "self = 500"
context [1] :
transactions->exist(points>0)
Exists
property [1..*] : transactions
constraints [1..*] : transaction points > 0
context [1] : LoyaltyAccount
transaction points > 0
AttributeValueRestriction
property [1..*] : points
operator [1] : ">"
value [1] : "0"
context [1] : Transaction

Figure 8.23: Pattern instances for loyalty accounts.

```
context ProgramPartner
inv servicePartner: ObjectInCollection(ProgramPartner,
                                        delivered Services.level .program.partners,
                                        Sequence{})
```

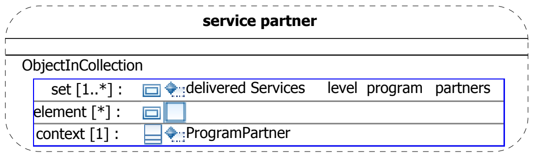

service partner
ObjectInCollection
set [1..*] : delivered Services level program partners
element [*] :
context [1] : ProgramPartner

Figure 8.24: Pattern instances for service levels.

8.4.3 Quantitative Evaluation.

In this subsection, we perform a quantitative evaluation of the patterns approach for the Royal & Loyal model. We use the following criteria as defined in Section 8.1: specification coverage, conciseness, performance, and elicitation coverage.

Specification Coverage.

From the 26 constraints for the Royal & Loyal model, we can express 18 using our existing constraint patterns, which corresponds to a ratio of about 70%. It requires 31 pattern instances to express these 18 constraints because many constraints require the use of composite patterns such as *If-Then-Else* or *Exists*. In Table 8.5, we summarize which patterns we have used and how often they occur.

This case study motivates one new constraint pattern because there are two constraints with similar structure for which there is currently no pattern. Constraints 8.4.17 and

Occurrences	Pattern
12	*Attribute Value Restriction*
6	*Multiplicity Restriction*
4	*If-Then-Else*
4	*Exists*
3	*Object In Collection*
1	*ForAll*
1	*Literal OCL*

Table 8.5: Pattern instances used for Royal & Loyal.

8.4.18 require unique attribute values for a given set of objects. Thus, we introduce a new pattern *Unique Set Identifier* and define it as follows.

```
pattern UniqueSetIdentifier (set:Sequence(Property), property:Tuple(Property)) =
  self.set−>isUnique(property)
```

Using this pattern, constraints 8.4.17 and 8.4.18 can be expressed as follows.

```
context LoyaltyProgram
inv uniqueAccount: UniqueSetIdentifier(LoyaltyProgram,Membership.account,number)
inv uniqueNames: UniqueSetIdentifier(LoyaltyProgram,participants,name)
```

By introducing one new pattern, we have increased the coverage from 18/26 to 20/26, which is about 77%.

Another constraint that could not be expressed using our patterns is constraints Constraint 8.4.20. However, we can introduce a new composite pattern *Exists One* that requires the existence of exactly one object in a given set S with given properties P. We define it in HOL-OCL as follows.

```
constdefs
  ExistsOne :: "((′τ, ′a::bot)VAL ⇒ ′τ Boolean)list ⇒ (′τ, ′a::bot)Set ⇒ ′τ Boolean"
  "ExistsOne P S == S−>one(y | (oclAND P y))"
```

Using this new pattern, Constraint 8.4.20 can be expressed as follows.

```
context LoyaltyProgram
inv oneAccount: ExistsOne(AttributeValueRestriction(LoyaltyAccount,number,<,10000),
                          self.Membership.account)
```

Still, a few constraints cannot be expressed using our constraint patterns. In Table 8.6, we present a list of constraints that we cannot express using constraint patterns and briefly explain why this is the case.

Conciseness.

For the constraints of the Royal & Loyal model, constraint patterns do not offer a solution that is substantially more concise than the corresponding OCL statements. This is caused by the fact that none of the constraints requires complex recursive queries such as transitive closure operators. Nevertheless, the constraint pattern offer an abstraction from the concrete OCL syntax, which may be considered an advantage by model developers with no OCL background.

Constraint(s)	Description
8.4.2	This constraint contains an expression in which the cardinalities of sets are compared.
8.4.4	This constraint (and also 8.4.5) comprises function calls, which is not supported by our approach.
8.4.13	This constraint uses association classes for navigation, which is currently not supported by our constraint patterns.
8.4.14	This constraint comprises statements about ordered sets.
8.4.22	This constraint comprises filtering of sets using the OCL select() operation.

Table 8.6: Constraints from the Royal & Loyal model not expressible using our patterns.

Analysis Performance.

Our consistency analysis for the Royal & Loyal model runs for about one second and results in 15 warnings. After a manual check, all warnings turn out to be false positives. Despite this fact, one of these warnings provides interesting insight and motivates a change in the model: The analysis issues a warning for the constraint *no account*, which restricts the multiplicity of account to zero. The warning reports that the multiplicity of account should be $*$ in the model. This would allow an arbitrary number of elements to participate in the relation, and thus, no instance of *Multiplicity Restriction* could potentially violate this multiplicity. However, the multiplicity range of account in the model is [0..1]. Although this is not a contradiction with *no account*, we suggest to remove the association end account in the model to make it more concise.

Elicitation Coverage.

Before adding the constraints to the model, constraint elicitation reports 26 warnings for the Royal & Loyal model. After adding the constraints, the same 26 warnings are still reported, which means that none of the added constraints removes any anti-pattern. Due to this relatively high number of warnings, we further investigated the warnings.

From all 26 warnings, 22 warnings address unconstrained reflexive associations, which are not restricted by any of the 26 constraints for the Royal & Loyal model. In fact, cycles are desired in this model: A navigation programs.partners.deliveredServices.transactions.card.owner starting at Customer should result in a set that contains the respective customer. Therefore, it is unlikely that any kinds of cycles should be excluded from the model.

Three of the remaining four warnings suggest using *Type Restriction* constraints for associations between classes and superclasses, e. g., Service and Transaction. For these warnings, the model developer must assess whether the possible types of the respective association ends should be restricted. Whereas customer cards should be associable to any kind of transaction, it may make sense to limit certain services such that they can be related to either "burning" or "earning" transactions.

The remaining warning suggests restricting the unbounded multiplicity of the association end cards of class Customer.

8.5 Qualitative Evaluation and Summary

We have performed three case studies and evaluated them according to *quantitative* measures. In this section, we apply the *qualitative* evaluation criteria defined in Section 8.1 and summarize the validation of our approach.

8.5.1 Analysis Quality.

Analyzing the constraints for the monitor model, the process-merging prototype, and the Royal & Loyal model resulted in warnings that turned out to be false positives. This means that in general, the assumptions used in the consistency theorems are too strong. In Section 9.2, we discuss how these assumptions can be weakened in order to reduce the number of false positives.

We have seen that consistency analysis results in 30 warnings for the constrained monitor model. As seen in Section 8.2.4, there is no automatic tool at the moment that can be used to perform a secondary consistency analysis. Thus, we needed to manually go through the list of warnings and decide for each warning whether it is an actual inconsistency or a false positive. As before, we (informally) created a model state that satisfies all pattern instances affected by the respective warning and thus provided a witness.

Although this is a time-consuming endeavor, we consider our pattern-based analysis successful. After all, 86 pattern instances can be shown consistent by our analysis and thus, we do not have to analyze them further. This is an improvement to the current state-of-the-art, because there are no consistency analysis methods for UML/OCL models that can handle models of this size and intricacy. As a consequence, all 71 constraints from the constraint specification must be analyzed *manually* in state-of-the-art MDE.

In the second case study, consistency analysis correctly identifies the Constraint 8.3.3 and Constraint 8.3.5 as inconsistent because they do not allow certain classes to be instantiated. As shown, the remaining warnings are false positives. However, these warnings helped identify redundancies in the constraint specification. We subsequently eliminated the redundancies, which made the constraint specification more concise.

For the Royal & Loyal model, consistency analysis reported 15 warnings, which all turned out to be false positives in a manual consistency analysis. However, the results from consistency analysis provided insights into the model and helped us to make the model more concise.

However, our analysis cannot make statements about arbitrary OCL constraints. This is one of the limitations of our approach, which we discuss in the following subsection.

8.5.2 General Insights.

In general, the case studies showed that a majority of constraints can be expressed using our library of composable constraint patterns. Around 75% of the constraints from the *monitor model* and the *Royal & Loyal* model can be expressed using constraint patterns and from the *process-merging prototype* case study, even 100% of the constraints can be “patternized”. In this subsection, we highlight general insights that we gained from the respective case study.

In the case study about the monitor model, the majority of constraints could be expressed using constraint patterns. Furthermore, the patterns approach provides a more concise view on the constraints, which promises to be easier to maintain for a majority of constraints. Using constraint patterns is an improvement for constraint maintenance. Upon model change, it is often sufficient to adapt the parameter values of the constraint pattern instances; in our prototype implementation in RSA (Chapter 7), renaming of model elements is automatically reflected in the constraint pattern instances. Another advantage of the patterns approach is that redundancy can be avoided. Using patterns, it is less likely to develop constraints that are syntactically different but semantically imply each other because we have noticed that such similarities become more obvious to the user in a pattern approach. An example for such similar constraints are constraints C.10.1 and C.10.5 from Appendix C.

In the case study on formalizing constraints on the input models of a process-merging prototype, all constraints for the models can be formalized using our library of constraint patterns. Although this case study is rather small, it provides insight into special kinds of constraints, i. e., restrictions on models that are caused by the temporary immaturity of model-processing software. In this case study, we have come across two types of constraints. First, certain kinds of classes are not allowed to be instantiated, i. e., no objects of this class may exist. Such constraints are typically used when certain classes in the meta-model denote special cases in the model's domain and the software that processes the models is not yet able to consume these special cases. In this study, input models must not contain objects of the classes TerminationNode and LoopNode. Second, certain associations in the meta-model may not be supported by the software prototype to their full extent and thus need to be further restricted. In this study, models are not allowed to have implicit control flow, which means that each StructuredActivityNode object may have at most one link to an OutputControlPin object although the meta-model is more liberal (cf. Figure 8.14). Besides the cardinality, the possible types of association ends may be restricted. In Table 8.7, we summarize typical use cases for constraints in early phases of development.

Use Case	Pattern(s)
A certain class in the model denotes a special case that is typically not implemented in the beginning of the development process.	*No Instances*
Links between objects are restricted either in their number or in the types of objects to be connected.	*Multiplicity Restriction* *Type Restriction*

Table 8.7: Typical constraint patterns in early development phases.

The third case study on the Royal & Loyal model confirmed the insights made in the previous case studies. In particular, we used the constraints for the Royal & Loyal model to elicit one new elementary constraint pattern, *Unique Set Identifier*, and one new composite constraint pattern, *Exists One*. This case study also comprised several constraints that cannot be expressed using our library. In the following list, we summarize constraint types that currently defy being expressed using our patterns.

- **Cardinalities of sets** can currently not be compared.
- **Function calls** are generally not supported in constraint expressions.

- **Association classes** cannot be used for navigation.
- **Ordered sets** cannot be constrained using our patterns.
- **Filtering of sets** according to their type is not supported.
- **Binding of variables** in quantified expressions is not flexible enough.

Whereas patterns may be introduced for expressing constraints on cardinalities, association classes, and ordered sets, it is an open question how filtering expressions and a flexible variable binding in quantified expressions can be supported using patterns. Functions calls are generally not supported by our approach. We discuss these limitations in the following subsection.

8.5.3 Limitations.

Our approach to developing constraint specifications using patterns has a few limitations that have become obvious during this evaluation. In particular, we have identified the following three limitations.

Arbitrary OCL Constraints.

There will always be constraints that state certain facts that are so specific to a certain domain that it is not feasible to capture them as patterns. As a result, these constraints remain as literal OCL constraints in the model. This has a severe impact on consistency analysis because our analysis cannot make statements about arbitrary OCL constraints.

We have shown a solution to this problem that requires significant user interaction: The model developer must manually create witnesses that satisfy all constraints in the model, including those pattern instances for which warnings are issued and all literal OCL constraints. In future extensions to our approach, the developer can be supported in providing witnesses by technologies such as those illustrated in Section 6.1.1.2. When such technologies have been developed, it is an open question whether such witness creation approaches complement our analysis approach or whether our analysis approach can be completely replaced by an automatic witness creation approach that uses knowledge about the constraint patterns for a more targeted witness generation. We further discuss this question in Section 9.2.

User-Defined Functions.

In the course of constraint development, it is often required to define recursive functions, which must be hand-written in OCL. Basically, there are two types of these functions: transitive closure operations and domain-specific parsing of recursive expressions. This problem could partly be overcome by providing patterns that represent recurring expressions, as we did for example with the transitive closure operation, which can be combined to define complex recursive functions. In particular, we have noticed that it is often necessary to define functions that aggregate specific metrics of a model. In fact, almost 66% of the constraints that cannot be expressed using patterns require user-defined functions. In future work, new means of defining such functions in a more concise and pattern-oriented way should be explored and ways of using such functions in MDE tools should be explored. For instance, the patterns framework in RSA is currently not flexible enough to define such user-defined functions in a useful way because in RSA, patterns do not have types. For example, this makes it impossible to replace a parameter value of type integer by a pattern instance that evaluates to integer. Thus, a more fine-grained approach is desirable in which patterns can represented any kind of (OCL) sub-expression.

Fixed Variable Binding.

The third limitation is that composite constraint patterns can be used in a limited way only since they do not support complex variable binding. This is especially a problem with the *ForAll* and *Exists* patterns. For instance, an expression self.x−>forAll(k:K | k.y = z) can be expressed using the patterns *ForAll* and *Attribute Value Restriction* pattern where x denotes an attribute of type K belonging to the context object, y denotes an attribute of objects k of class K, and z denotes an arbitrary term. However, since the structure of the *ForAll* pattern is static, it is not possible to reference the object self in the term z. For the case that the association between self and x is one-to-one, this can be overcome by a navigation expression that navigates from the quantified object k to self.

We illustrate this limitation in terms of the following constraints for the company model in Section 2.2.1.

```
context Manager
inv salary: self.employs−>forAll(e | e.salary = 3000)
inv office : self.employs−>forAll(e | e.worksIn = self.worksIn)
```

Whereas the first constraint *salary* can be expressed using our constraint patterns, the second constraint *office* cannot. This becomes clear when we formalize *salary* using constraint patterns.

```
context Manager
inv salary: ForAll(Manager,employs,AttributeValueRestriction(Employee,salary,=,3000))
```

The problem with expressing the second constraint using patterns is that the quantified statement refers to the context object self. Consider the following formalization.

```
context Manager
inv office : ForAll(Manager,employs,AttributeValueRestriction(Employee,worksIn,=,
                                                              worksFor.worksIn))
```

Although this pattern may look correct at first glance, it is *not*: The expression worksFor.worksIn evaluates to the set of the offices of all managers of some employee. However, if the association between Employee and Manager was one-to-one, the constraint could be expressed as shown above.

8.5.4 Summary.

The case studies have shown that the approach developed in this thesis can effectively be applied in real-world development projects. We were able to formalize a majority of constraints using our patterns and it has turned out that our consistency analysis makes reasonable statements about a subset of the constraints.

However, it has also turned out that around 25% of constraints cannot be expressed using our patterns, and consistency analysis produces numerous false positives, i. e., it warns that consistent constraints may be inconsistent. Despite these limitations, we think that our approach is a step towards the development of concise and consistent constraint specifications and furthermore, it can be extended to offer a higher expressiveness and a lower rate of false positives.

Chapter 9

Conclusion

In this thesis, we have presented an MDE process that provides a method and tools for the development of concise and consistent constraint specifications. The focus of this process is practicability, which we obtain by capturing knowledge as patterns and use this knowledge for automating several tasks.

In this chapter, we summarize the contributions made in this thesis in Section 9.1, give an outlook on future work in Section 9.2, and report on lessons learned in Section 9.3.

9.1 Contribution Summary

In this section, we summarize the contributions made in this thesis and explain why we consider each contribution important.

Constraint Elicitation and Specification.

We identified a set of anti-patterns that occur in unconstrained class models and illustrated typical problems caused by these anti-patterns. Consequently, we explained how these anti-patterns can be remedied by enriching class models with textual constraints.

To the best of our knowledge, this is the first approach to automatically eliciting constraints for class models that complements classical domain analysis. Current literature covers the elicitation and specification of constraints by example [Kleppe and Warmer, 2003] only and does not provide a guided way of eliciting and specifying constraints. Not using a guided approach increases the probability of specifying models insufficiently, which can cause undesirable effects in the generated code. Furthermore, eliciting certain constraints, e. g., on reflexive associations, requires a thorough understanding of the intricacies of object-oriented modeling. Since model developers are typically domain experts without fundamental knowledge in formal languages, they can be significantly supported by an automatic search for occurrences of anti-patterns.

Specifying Constraints Using Patterns.

We presented an approach to creating concise constraint specifications for class models using composable constraint patterns, which represent a considerable improvement in expressiveness compared to previous constraint-pattern approaches. We showed how the semantics of constraint patterns can be precisely defined in terms of functions in higher-order logics. Consequently, we introduced an extensible library of constraint patterns that cover typical specification tasks, in particular remedying the previously identified anti-patterns.

Although MDE is already successfully used in industry projects and OCL has been around for several years, refinements are typically not applied during the design phase of the development process, but only in the code that has been generated from the class model. The reason for this is that developing constraints in OCL is typically not performed by some model developer, who is an expert in the domain of the model, but by some constraint developer, who is an expert in formal specification languages. However, the role of constraint developer is typically not occupied in industry projects, which is why model developers are often responsible for creating formal constraint specifications. Our approach of composable constraint patterns allows model developers to develop constraints without requiring knowledge in formal languages. Formalizing the constraint patterns in HOL-OCL allowed us to increase the degree of automation in consistency proofs in the subsequent chapter.

Consistency of Constraint Specifications.

We assembled several consistency notions for object-oriented specifications from the literature and discussed their practical relevance for constrained class models. We presented how proof obligations for interactive theorem proving can be generated for the different consistency notions and how these obligations can be proven. We further showed how constraint patterns can be used to increase the degree of automation in interactive consistency proofs.

There are numerous publications about consistency of models, and various MDE tools comprise consistency analyses. However, it is often unclear in state-of-the-art tools and publications what the exact semantics is for consistency in the respective context. Therefore, we consider Chapter 5 an important contribution to clarify these notions and suggest practically relevant ones. We also presented that constraint patterns can be useful in interactive consistency analysis. In the subsequent chapter, we showed how they can also be useful in automatic consistency analysis.

Consistent Model Refinement Using Patterns.

We introduced a heuristic approach for analyzing the consistency of pattern-based constraint specifications with polynomial complexity. This approach integrates seamlessly into the MDE process and can be complemented by existing analysis approaches.

There are several approaches and tools for analyzing the consistency of UML/OCL models. However, they have turned out not be useful for large real-word models because they either support only a subset of UML/OCL or choke on the size of the model to be analyzed. Therefore, we consider our approach an important contribution because it can efficiently analyze even large models, make consistency statements for a majority of constraints, and runs completely automatically.

Tool Support and Validation.

We developed a set of plug-ins that allow model developers to effectively use the approach introduced in this thesis in a commercial development tool. Furthermore, we validated our approach and implicitly assessed the usability of the plug-ins by performing several case studies on nontrivial models.

Since the unique selling proposition of MDE is the automation of various time-consuming tasks, proper tool support is crucial for the success of MDE. The plug-ins that we have developed support model developers in eliciting constraints through an automatic analysis, specify constraints in a concise way using constraint patterns, analyze pattern-based constraint specifications for consistency, and generate OCL code from such specifications. In our validation, we could show that a majority of practically relevant constraints can be expressed with our approach and that our consistency analysis is the only analysis available for UML/OCL that is feasible for analyzing large and complex industry models.

9.2 Future Work

We have identified the anti-patterns discussed in Section 3.1 as the most important ones for increasing the maturity level of class models. Future work can extend our findings by investigating further specification problems that frequently occur in the MDE process, both independent of the application domain and specific to certain domains.

We have introduced the concept of composable constraint patterns and presented an extensible library of patterns. Future work can extend this library by adding constraint patterns that have been shown useful in practice. For example, we have elicited the *No Instances* pattern in the case study in Section 8.2.

We have defined several distinct notions of consistency for constrained class models. We see two directions for future work on this topic. First, a formal notion of compositionality can be defined based on our consistency definitions. For example, for showing class consistency of a class-consistent model extended by a new specialized class, it is sufficient to show weak consistency for the new class and subtype consistency for its specialization relation. Second, we have seen in the case study in Section 8.3 that for early development phases, weak consistency may be an appropriate notion of consistency. Future work can investigate details about different notions of consistency for different phases of the development process. Furthermore, future work comprises communicating inconsistencies to the model developer in an efficient way. It must be pointed out what the exact cause of an inconsistency is and a link must be given to the appropriate place in the model or in the constraint specification. Furthermore, it needs to be investigated how the user can be given hints on how to remedy inconsistencies.

We have discussed only briefly how our approach integrates into an MDE process. There are several directions for future work here. First, it can be investigated how our approach can be used in distributed work environments where several model developers collaborate on writing constraint specifications. Second, it can be researched how the constraints can be integrated into the generated program code and evaluated at runtime. For example, the pattern instances may be transformed into Java code, which motivates efficiency considerations on the generated code. Third, it can be worked out in further detail what relevant consistency notions are in different (in particular, early) phases of development. Since our case studies cover “mature” models only, one may gain insights by performing further case studies in which models are incrementally created and constraints incrementally added.

The heuristic analysis based on patterns as introduced in this chapter can be improved in three ways. First, the assumptions in the consistency theorems should be weakened in a sensible way in order to reduce the fraction of cases in which the consistency analysis returns “Don’t know.” For example, inequality reasoning can be used for patterns in which numeric values of attributes or the multiplicities of association ends are important. For instance, assumption (iv) in Theorem 16 can be improved by analyzing the parameters of two instances of the *Multiplicity Restriction* pattern that restrict the same association end. The fraction of cases in which the consistency analysis returns “Don’t know” can further be reduced by establishing a second theorem for each constraint pattern. Whereas the theorems stated in Section 6.3 describe the assumptions under which a pattern instance *preserves* the consistency of a model, theorems can be stated about the assumptions under which a pattern instance is certain to *violate* the consistency of a model. Using such theorems, each pattern instance can be checked for whether its consistency assumptions hold, its inconsistency assumptions hold, or if neither assumptions hold. This adds one more possible result to our analysis: besides “consistent” and “don’t know”, the analysis can also return “inconsistent” for a given constraint specification.

Second, our approach can be complemented by interactive witness generation as described in Section 6.1.1.2. Whenever a pattern instance cannot be shown consistent according to the consistency assumptions, the user is asked to provide a state in which the classes constrained by the pattern instance are represented and that satisfies the constraint specified by the pattern instance. Such an approach replaces secondary consistency analysis by approaches such as SAT-based analysis or interactive theorem proving, who have shown not yet to be suitable for real-world models.

Third, future work should define a process for adapting the consistency theorems for the constraint patterns when the library of constraint patterns is extended. In Section 6.5.2, we have already stated that on the one hand, such a process comprises stating a new consistency theorem for each pattern that is added to the library and on the other hand, the consistency theorems for the existing constraint patterns must be checked whether they still hold and adapted if needed.

However, even with these improvements, our analysis approach has one major disadvantage: it cannot make statements about arbitrary OCL constraints, i. e., constraints that cannot be expressed by constraint patterns. We have also seen that current approaches to automatic witness creation have a problem with scaling to large models and constraint specifications. Thus, future work could enhance witness creation approaches by augmenting them with pattern information: Similar to the consistency theorems that capture generic relationships between patterns, knowledge about patterns can be explored that provide a heuristic for avoiding exponential growth in witness creation.

Future work can also comprise enhancing the tool support in several ways. First, pattern-mining techniques could be used to map existing OCL constraints to our patterns and thus incorporate existing constraints in the analysis. Second, more model transformations can be added to COPACABANA to support further target languages, e. g., Java, Eiffel, SQL, or XQuery. Third, experts in GUI design can improve the user interface. In particular, pattern instances could be hidden from the user and replaced by other elements in the model that can be used more intuitively by the model developer, e. g., stereotypes.

Besides consistency, an interesting property of constraint specifications is redundancy. Redundancy in constraint specifications causes two problems. First, when the model is changed during maintenance, redundant constraints can lead to inconsistencies if not all redundant constraints are changed accordingly. Second, redundant constraints compromises the performance of evaluating a given model state against a constraint specification because identical calculations are carried out more than once. Analyzing pattern-based constraint specifications for redundancy can be performed in a similar way than analyzing them for consistency: Instead of specifying theorems that state relationships between patterns with respect to consistency, new theorems can be established that state relationships between patterns with respect to redundancy. Thus, redundancies between pattern instances can be detected efficiently.

9.3 Final Remarks

The contributions made in this thesis are in the area of Model-Driven Engineering (MDE), a promising new approach to software engineering. In this section, we report on several lessons learned about MDE in the course of writing this thesis.

Whereas MDE is generally considered to reduce the complexity in software development, we object to this view. MDE does not *reduce* the complexity inherent to most systems, but we are convinced that MDE provides effective means for *dealing* with this complexity by allowing model developers to represent different aspects of the system with different model types. Thus, the model developer can choose the most suitable languages for representing the respective aspects of the system.

We have noticed that modeling languages are often too complex, and as a result, model developers are likely to make similar mistakes as with general-purpose programming languages. It is important to follow the concept of separation of concerns and not mix different aspects of a system in the same modeling language. In contrast, distinct modeling languages, often called domain-specific languages (DSL) together with model transformations that can integrate the different aspects promise to increase the acceptance of MDE. We consider the development of modeling languages and related model transformations an important research avenue, and this is confirmed by a growing interest in DSLs both from research and industry.

Although design patterns are well-established and have become common sense in software engineering, the full potential of pattern-based approaches has not yet been unleashed. Patterns provide predefined solutions with user-friendly interfaces for applying best practices to various problems. Constraint patterns as we have used them in this thesis are one example for applying the pattern approach to a very specific domain. We are convinced that patterns will play an even more important role in the future of MDE for modeling various aspects of systems.

We were surprised to find very little about MDE methodology, and existing literature is often vague. For example, the terms *analysis model* and *design model* are often distinguished in the literature, but it is unclear which levels of abstraction are appropriate for each model *exactly*. In our opinion, numerous methodological aspects of MDE are unclear, which hinders a broad acceptance of MDE in software engineering development processes. Thus, we consider both theoretical and practical research on MDE methodology an important necessity.

As mentioned before, tool support is crucial for the success of MDE. However, we have experienced numerous deficiencies when working with available MDE tools, which leaves room for improvement for researchers and tool developers. In the following, we enumerate several of these deficiencies: Tools often support only certain subsets of languages, as we experienced with tools for consistency analysis. This qualifies these tools as playgrounds for toy models, but disqualifies them for real-world models such as those in our case studies. In additions, existing tools often do a bad job in hiding the complexity of the modeled system from the user and as mentioned before, modeling languages can be complex. A laudable exception is for example IBM WebSphere Business *Modeler* (WBM), which offers different views (basic, medium, advanced) on a business process, which helps users focus on either business aspects or technical aspects of process models. In contrast, tools hide important information regarding the semantics of the model that would be important to the model developer. For example, we have come across tools for consistency analysis that do not contain any hint about what notion of consistency they actually evaluate.

Bibliography

[Abrial and Mussat, 2002] Abrial, J.-R. and Mussat, L. (2002). On using conditional definitions in formal theories. In *ZB'2002 - Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 242–269, Grenoble, France.

[Ackermann and Turowski, 2006] Ackermann, J. and Turowski, K. (2006). A Library of OCL Specification Patterns to Simplify Behavioral Specification of Software Components. In *Proceedings of Conference on Advanced Information Systems Engineering.*, number 4001 in LNCS, pages 255–269.

[Ahrendt et al., 2005] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P. H. (2005). The KeY Tool. *Software and System Modeling*, 4(1):32–54.

[Aichernig and Larsenz, 1997] Aichernig, B. K. and Larsenz, P. G. (1997). A Proof Obligation Generator for VDM-SL. In *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 338–357.

[Akehurst and Patrascoiu, 2004] Akehurst, D. H. and Patrascoiu, O. (2004). OCL 2.0 - Implementing the Standard for Multiple Metamodels. *Electronic Notes in Theoretical Computer Science*, 102:21–41.

[Álvarez et al., 2003] Álvarez, J. A. T., Requena, V., and Fernández, J. L. (2003). Emerging OCL Tools. *Software and System Modeling*, 2(4):248–261.

[Amálio et al., 2006] Amálio, N., Stepney, S., and Polack, F. (2006). A Formal Template Language Enabling Metaproof. In *FM 2006: Formal Methods*, volume 4085 of *LNCS*.

[Atkinson and Kühne, 2003] Atkinson, C. and Kühne, T. (2003). Model-driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41.

[Baar, 2003] Baar, T. (2003). The Definition of Transitive Closure with OCL – Limitations and Applications. In *Proceedings, Fifth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, volume 2890 of *LNCS*, pages 358–365. Springer.

[Basin et al., 2006] Basin, D., Doser, J., and Lodderstedt, T. (2006). Model Driven Security: from UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91. An intermediate version is available as ETH technical report no. 414.

[Beckert et al., 2002] Beckert, B., Keller, U., and Schmitt, P. H. (2002). Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC)*, pages 113–123, Copenhagen, Denmark.

[Behm et al., 1998] Behm, P., Burdy, L., and Meynadier, J.-M. (1998). Well Defined B. In *B98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 29–45.

[Berardi et al., 2005] Berardi, D., Calvanese, D., and De Giacomo, G. (2005). Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168(1):70–118.

[Bird and Wadler, 1988] Bird, R. J. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.

[Bordbar and Anastasakis, 2005] Bordbar, B. and Anastasakis, K. (2005). UML2Alloy: A Tool for Lightweight Modelling of Discrete Event Systems. In *Proceedings of IADIS International Conference in Applied Computing 2005*, pages 209–216, Algarve, Portugal.

[Bottoni et al., 2000] Bottoni, P., Koch, M., Parisi-Presicce, F., and Taentzer, G. (2000). Consistency Checking and Visualization of OCL Constraints. In Evans, A., Kent, S., and Selic, B., editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 294–308. Springer.

[Boulton et al., 1992] Boulton, R., Gordon, A., Gordon, M. J. C., Harrison, J., Herbert, J., and Tassel, J. V. (1992). Experience with Embedding Hardware Description Languages in HOL. In *International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume 10, pages 129–156.

[Brucker, 2007] Brucker, A. D. (2007). *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Switzerland.

[Brucker et al., 2006] Brucker, A. D., Doser, J., and Wolff, B. (2006). Semantic Issues of OCL: Past, Present, and Future. In *Proceedings of the 6th OCL Workshop at the UML/MoDELS Conference 2006*, pages 213–228.

[Brucker and Wolff, 2006] Brucker, A. D. and Wolff, B. (2006). The HOL-OCL Book. Technical Report 525, ETH Zurich, Switzerland.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented Software Architecture: a System of Patterns*. John Wiley & Sons, Inc. New York, NY, USA.

[Cabot, 2006] Cabot, J. (2006). Ambiguity Issues in OCL Postconditions. In *Proceedings of the 6th OCL Workshop at the UML/MoDELS Conference 2006*, pages 194–204.

[Chen et al., 2006] Chen, S.-K., Lei, H., Wahler, M., Chang, H., Bhaskaran, K., and Frank, J. H. (2006). A Model Driven XML Transformation Framework for Business Performance Management Model Creation. In *International Journal of Electronic Business*, volume 4, pages 281–301. Inderscience.

[Chiorean et al., 2005] Chiorean, D., Bortes, M., and Corutiu, D. (2005). Proposals for a Widespread Use of OCL. In Baar, T., editor, *Proceedings of the MoDELS'05 Conference*

Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, Lausanne, Switzerland.

[Chiorean et al., 2004] Chiorean, D., Corutiu, D., Bortes, M., and Chiorean, I. (2004). Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *Proceedings of NWUML'2004 – 2nd Nordic Workshop on the Unified Modeling Language*, pages 127–142.

[Chiorean et al., 2003] Chiorean, D., Paşca, M., Cârcu, A., Botiza, C., and Moldovan, S. (2003). Ensuring UML Models Consistency Using the OCL Environment. In *UML 2003 - Workshop: OCL 2.0 - Industry standard or scientific playground?*

[Clavel and Egea, 2006] Clavel, M. and Egea, M. (2006). ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 4019 of *LNCS*, pages 368–373, Kuressaare, Estonia.

[Correa and Werner, 2004] Correa, A. L. and Werner, C. M. L. (2004). Applying Refactoring Techniques to UML/OCL Models. In Baar, T., Strohmeier, A., Moreira, A. M. D., and Mellor, S. J., editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer.

[Costal et al., 2006] Costal, D., Gómez, C., Queralt, A., Raventós, R., and Teniente, E. (2006). Facilitating the Definition of General Constraints in UML. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, *MoDELS 2006*, number 4199 in LNCS, pages 260–274. Springer-Verlag.

[Cox, 1986] Cox, B. J. (1986). *Object-oriented Programming : an Evolutionary Approach*. Addison-Wesley.

[Cranefield and Purvis, 1999] Cranefield, S. and Purvis, M. (1999). UML as an Ontology Modelling Language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*.

[Damm et al., 1991] Damm, F. M., Hansen, B. S., and Bruun, H. (1991). On Type Checking in VDM and Related Consistency Issues. In *4th International Symposium of VDM Europe on Formal Software Development - Volume I*, volume 551 of *LNCS*, pages 45–62.

[Darvas and Müller, 2006] Darvas, A. and Müller, P. (2006). Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology*, 5:59–85.

[Demuth and Hussmann, 1999] Demuth, B. and Hussmann, H. (1999). Using UML/OCL Constraints for Relational Database Design. In France, R. B. and Rumpe, B., editors, *UML*, volume 1723 of *LNCS*, pages 598–613. Springer.

[Distefano et al., 2000] Distefano, D., Katoen, J.-P., and Rensink, A. (2000). Towards Model Checking OCL. In *Proceedings of the ECOOP Workshop on Defining a Precise Semantics for UML*.

[Dresden Technical University, 2007] Dresden Technical University (2007). Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net.

[Dwyer et al., 1998] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property Specification Patterns for Finite-state Verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA. ACM Press.

[Dzidek et al., 2005] Dzidek, W., Briand, L., and Labiche, Y. (2005). Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. *LNCS*, 3844:10–19.

[Eclipse Foundation, 2007a] Eclipse Foundation (2007a). Eclipse Model Development Tools. http://www.eclipse.org/modeling/mdt/.

[Eclipse Foundation, 2007b] Eclipse Foundation (2007b). Eclipse Platform. http://www.eclipse.org.

[Eclipse Foundation, 2007c] Eclipse Foundation (2007c). The Eclipse Modeling Framework. http://www.eclipse.org/emf.

[Elaasar et al., 2006] Elaasar, M., Briand, L. C., and Labiche, Y. (2006). A Metamodeling Approach to Pattern Specification. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 484–498. Springer.

[Emmerich et al., 2003] Emmerich, W., Finkelstein, A., and Nentwich, C. (2003). Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering*, pages 455–464, Portland, Oregon. IEEE Computer Society.

[Fowler, 1997] Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional.

[Frank, 2007] Frank, J. H. (2007). Introduction to Monitor Modeling. Online article. http://www-128.ibm.com/developerworks/architecture/library/ar-bam1/.

[Fraunhofer Fokus, 2007] Fraunhofer Fokus (2007). OSLO. Open Source Library for OCL. http://oslo-project.berlios.de/.

[Gallier, 1986] Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Gogolla et al., 2005] Gogolla, M., Bohling, J., and Richters, M. (2005). Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 4(4):386–398.

[Gogolla and Richters, 2002] Gogolla, M. and Richters, M. (2002). Expressing UML Class Diagrams Properties with OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK. Springer-Verlag.

[Gornik, 2002] Gornik, D. (2002). UML Data Modeling Profile. Technical Report 5, Rational Corp., Whitepaper TP162.

[Hauser and Koehler, 2004] Hauser, R. and Koehler, J. (2004). Compiling Process Graphs into Executable Code. In *Third International Conference on Generative Programming and Component Engineering*, number 3286 in LNCS, pages 317–336. Springer.

[Hauser et al., 2005] Hauser, R., Koehler, J., Sendall, S., and Wahler, M. (2005). Declarative Techniques for Model-Driven Business Process Integration. *IBM Systems Journal*, 44(1):47–65.

[Hjørland and Albrechtsen, 1995] Hjørland, B. and Albrechtsen, H. (1995). Toward a New Horizon in Information Science: Domain-Analysis. *J. Am. Soc. Inf. Sci.*, 46(6):400–425.

[Horn, 1992] Horn, B. (1992). Constraint Patterns as a Basis for Object Oriented Programming. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 218–233, New York, NY, USA. ACM.

[Huet et al., 2008] Huet, G., Kahn, G., and Paulin-Mohring, C. (2008). The Coq Proof Assistant. A Tutorial. Online article. http://pauillac.inria.fr/coq/V8.1/tutorial.html.

[IBM, 2007a] IBM (2007a). developerWorks. http://www.ibm.com/developerworks/.

[IBM, 2007b] IBM (2007b). WebSphere Business Modeler. http://www-306.ibm.com/software/integration/wbimodeler/.

[IBM, 2007c] IBM (2007c). WebSphere Business Monitor. http://www.ibm.com/software/integration/wbimonitor/.

[Jackson, 2000] Jackson, D. (2000). Automating First-order Relational Logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA. ACM Press.

[Jackson, 2002] Jackson, D. (2002). Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

[Jackson et al., 2000] Jackson, D., Schechter, I., and Shlyakhter, I. (2000). Alcoa: The Alloy Constraint Analyzer. *Proceedings of the International Conference on Software Engineering*, pages 730–733.

[Jonckers et al., 2003] Jonckers, V., Mens, T., Simmonds, J., and Van Der Straeten, R. (2003). Using Description Logic to Maintain Consistency between UML Models. In Stevens, P., Whittle, J., and Booch, G., editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer.

[Jones, 1990] Jones, C. B. (1990). *Systematic Software Development using VDM*. Prentice Hall. ISBN 0-13-880733-7.

[Kaneiwa and Satoh, 2006] Kaneiwa, K. and Satoh, K. (2006). Consistency Checking Algorithms for Restricted UML Class Diagrams. In Dix, J. and Hegner, S. J., editors, *FoIKS*, volume 3861 of *Lecture Notes in Computer Science*, pages 219–239. Springer.

[Kent, 2002] Kent, S. (2002). Model Driven Engineering. In Butler, M. J., Petre, L., and Sere, K., editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer.

[Kleppe and Warmer, 2003] Kleppe, A. and Warmer, J. (2003). *The Object Constraint Language. Second Edition.* Addison-Wesley.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

[Küster, 2004] Küster, J. (2004). *Consistency Management of Object-Oriented Behavioral Models*. Dissertation, University of Paderborn.

[Kyas et al., 2005] Kyas, M., Fecher, H., de Boer, F., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., and Kugler, H. (2005). Formalizing UML Models and OCL Constraints in PVS. *Electronic Notes in Theoretical Computer Science*, 115:39–47.

[Leavens et al., 2006] Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38.

[Lenzerini and Nobili, 1990] Lenzerini, M. and Nobili, P. (1990). On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. *Information Systems*, 15(4):453–461.

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841.

[Lodderstedt et al., 2002] Lodderstedt, T., Basin, D. A., and Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jézéquel, J.-M., Hußmann, H., and Cook, S., editors, *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer.

[Maraee and Balaban, 2007] Maraee, A. and Balaban, M. (2007). Efficient Reasoning about Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In Akehurst, D. H., Vogel, R., and Paige, R. F., editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer.

[Markovic and Baar, 2005] Markovic, S. and Baar, T. (2005). Refactoring OCL Annotated UML Class Diagrams. In *Lionel C. Briand and Clay Williams, editors. Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *LNCS*, pages 280–294.

[Martin, 1982] Martin, J. (1982). *Application Development without Programmers*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Martin, 1998] Martin, R. C. (1998). Java and C++: A Critical Comparison. *Java Gems: Jewels from Java Report*, pages 51–68.

[Meyer, 1992] Meyer, B. (1992). Applying 'Design by Contract'. *Computer*, 25(10):40–51.

[Miliauskaitė and Nemuraitė, 2005] Miliauskaitė, E. and Nemuraitė, L. (2005). Representation of Integrity Constraints in Conceptual Models. *Information Technology and Control*, 34(4):355–365.

[Muller, 1999] Muller, R. (1999). *Database Design for Smarties: Using UML for Data Modeling*. Morgan Kaufmann.

[Newman, 1942] Newman, M. H. A. (1942). On Theories with a Combinatorial Definition of “Equivalence”. *The Annals of Mathematics*, 43(2):223–243.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture notes in computer science. Springer-Verlag Berlin Heidelberg New York.

[Object Management Group (OMG), 2003] Object Management Group (OMG) (2003). UML 2.0 OCL Final Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf.

[Object Management Group (OMG), 2006a] Object Management Group (OMG) (2006a). Meta Object Facility (MOF) Core Specification. Version 2.0. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[Object Management Group (OMG), 2006b] Object Management Group (OMG) (2006b). Object Constraint Language. OMG Available Specification. Version 2.0. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[Object Management Group (OMG), 2006c] Object Management Group (OMG) (2006c). Unified Modeling Language: Superstructure. Version 2.1. OMG document ptc/06-04-02. http://www.omg.org/cgi-bin/doc?ptc/2006-04-02.

[Owre et al., 1996] Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. (1996). PVS: Combining Specification, Proof Checking, and Model Checking. *Computer-Aided Verification, CAV*, 96:411–414.

[Pretschner and Prenninger, 2005] Pretschner, A. and Prenninger, W. (2005). Computing Refactorings of Behavior Models. In Briand, L. C. and Williams, C., editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 126–141. Springer.

[Prieto-Díaz, 1990] Prieto-Díaz, R. (1990). Domain Analysis: An Introduction. *SIGSOFT Softw. Eng. Notes*, 15(2):47–54.

[Queralt and Teniente, 2006] Queralt, A. and Teniente, E. (2006). Reasoning on UML Class Diagrams with OCL Constraints. In *Proceedings of the 25th International Conference on Conceptual Modeling (ER 2006)*, volume 4215 of *LNCS*, pages 497–512. Springer.

[Reif et al., 2001] Reif, W., Schellhorn, G., and Thums, A. (2001). Flaw Detection in Formal Specifications. In *Automated Reasoning : First International Joint Conference, IJCAR*, volume 2081 of *LNCS*, pages 642–657, Siena, Italy.

[Reiter and Criscuolo, 1981] Reiter, R. and Criscuolo, G. (1981). On Interacting Defaults. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI’81)*, pages 94–100.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented Modeling and Design*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Ryndina, 2005] Ryndina, K. (2005). Improving Requirements Engineering: An Enhanced Requirements Modelling and Analysis Method. Master’s thesis, University of Cape Town, South Africa.

[Sabetzadeh et al., 2007] Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., and Chechik, M. (2007). Consistency Checking of Conceptual Models via Model Merging. In *Proceedings of the 15th IEEE International Requirements Engineering Conference 2007*.

[Saxon and Plette, 1962] Saxon, J. A. and Plette, W. S. (1962). *Programming the IBM 1401: A Self-instructional Programmed Manual*. Prentice-Hall.

[Seidewitz, 2003] Seidewitz, E. (2003). What Models Mean. *IEEE Software*, 20(5):26–32.

[Süß, 2006] Süß, J. G. (2006). Sugar for OCL. In *Proceedings of the 6th OCL Workshop at the UML/MoDELS Conference 2006*, pages 240–251.

[Tedjasukmana, 2006] Tedjasukmana, V. N. (2006). Translation of OCL Invariants into SQL:99 Integrity Constraints. Master's thesis, Technical University of Hamburg, Germany.

[van Dalen, 1997] van Dalen, D. (1997). *Logic and Structure*. Springer, third edition.

[van der Aalst et al., 2003] van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. In *Distributed and Parallel Databases*, volume 14, pages 5–51. Springer.

[Wahler, 2008] Wahler, M. (2008). *Model-Driven Software Development: Integrating Quality Assurance*, chapter A Pattern Approach to Increasing the Maturity Level of Class Models. Idea Group Inc. To appear.

[Wahler et al., 2007] Wahler, M., Koehler, J., and Brucker, A. D. (2007). Model-Driven Constraint Engineering. *Electronic Communications of the EASST*, 5.

[Walsh, 2003] Walsh, T. (2003). Constraint Patterns. In *Proceedings of Principles and Practice of Constraint Programming-CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland*, pages 53–64. Springer.

Appendix A

Additional Theorems and Proofs

A.1 Some Simple Equivalences

In this appendix, we show formalizations in Isabelle of some equivalences that we used in Section 8.2.3. Isabelle can prove all statements using simplification.

We used the following equivalence in Section 8.2.3 for defining constraint *history range context*.

```
lemma "(x ∈⋃A) = (∃a ∈A. x ∈ a)"
apply simp
done
```

We used the following equivalence in Section 8.2.3 for defining constraint *counter context*.

```
lemma "(∀s ∈⋃A. P s) = (∀a ∈A. (∀s ∈a. P s))"
apply simp
done
```

We used the following equivalence in Section 8.2.3 for defining constraint *correlation value*.

```
lemma "x ∈S = (∃y ∈S. x = y)"
apply simp
done
```

A.2 Diamond Configurations and the *Unique Path* Pattern

In this section, we show that the *Unique Path* pattern excludes diamond configurations in object states. Furthermore, we show that it only excludes diamond configurations.

Theorem 22. *Let ϕ be an instance of the* Unique Path *pattern with class C as context and a navigation path $P = p_1.\dots.p_n$. If $\tau \models \phi$, then τ does not contain diamond configurations between objects of class C and objects of class $type(p_n)$.*

Proof. We proof this theorem by contradiction. Assume a state τ, $\tau \models \phi$, $o_1, o_4 \in \tau$ and a diamond configuration between o_1 and o_4, i. e., there exist two distinct objects $o_2, o_3 \in \tau$ and links $(o_1, o_2), (o_1, o_3), (o_2, \dots, o_4), (o_3, \dots, o_4) \in \tau$ (cf. Definition 14). Thus, the intersection of objects reachable from o_2 and o_3 via P is not empty. Therefore, $o_2 = o_3$

because $\tau \models \phi$. This is a contradiction because we assumed o_2 and o_3 to be distinct objects. □

Theorem 23. *Let Φ be a constraint specification and τ be a state such that $\tau \models \Phi$. If there does not exist a diamond configuration between objects of class C and objects of class $type(p_n)$ in τ, then $\tau \models \psi$ where ψ is an instance of the* Unique Path *pattern with class C as context and a navigation path $P = p_1. \dots .p_n$.*

Proof. Since there are no diamond configurations in τ between objects of class C and objects of class $type(p_n)$ in τ, it holds for all objects o_2, o_3 for which there exist links (o_1, o_2) and (o_1, o_3), there is no object o_4 that can be reached via path P from o_2 and o_3. Thus, the assumption of the implication in ψ is false, which makes ψ true and thus, $\tau \models \psi$. □

Appendix B

Implementation Details of COPACABANA

In this appendix, we elaborate on important implementation details of COPACABANA for two reasons. First, we want to give insights into the API that helps programmers to extend COPACABANA in the future. Second, understanding the details helps to develop solutions for similar problems in a model-driven way.

B.1 Constraint Elicitation

This component comprises four basic classes. The class AnalyzeModelAction adds an item to the context menu of a model in RSA from which the constraint elicitation can be invoked. The actual elicitation is performed by ClassModelAnalysis, which analyzes the model on which it has been invoked for the problems described in Section 3.1. Each result of the elicitation is represented as an object of class AnalysisResult, which stores properties such as the context and the description of the result. The results are displayed in ClassModelAnalysisView, a specialization of the ViewPart class, which is provided by Eclipse. Figure B.1 shows details of these classes as class diagram.

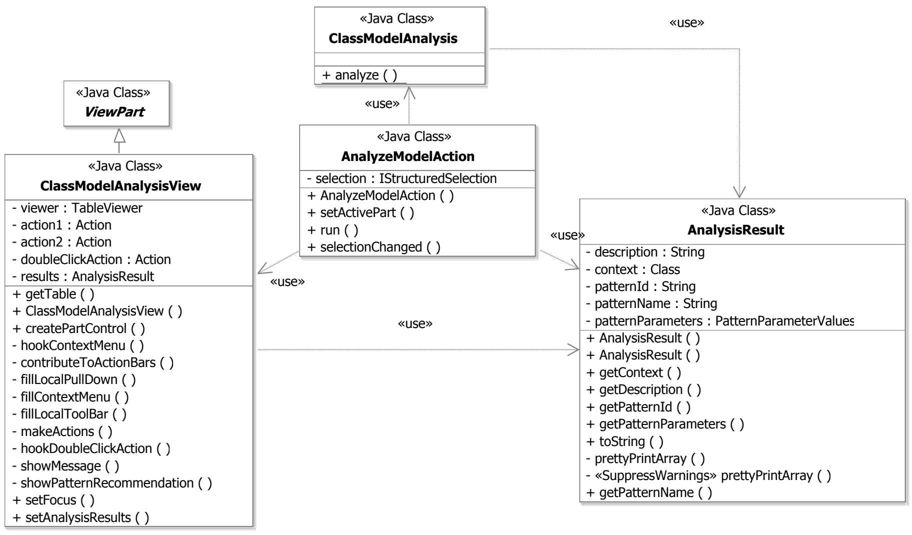


Figure B.1: Class diagram of the constraint elicitation component.

The following listing shows an excerpt of the analyze() method in class ClassModelAnalysis, which is invoked by the constraint elicitation component for each class in a given model. In this code snippet, constraint elicitation checks whether the *Missing Unique Identification* pattern holds, and if it does, it adds a new warning to the list of results.

```
public static List<AnalysisResult> analyze(Object object) {
  List<AnalysisResult> result = new ArrayList<AnalysisResult>();

  if (object instanceof Class) {
    Class klass = (Class) object;

    /* hasUniqueProperty determines whether a class has at least one property that is unique.
     * initially , we set it to false, and we will set it to true once we find a unique
     * property
     */
    boolean hasUniqueProperty = false;

    /*
     * analyzing the properties of a class
     */
    for ( Iterator iterator = klass. getAllAttributes ()
        . iterator (); iterator .hasNext();) {
      Property property = (Property) iterator .next ();
      if (UML2Helper.propertyIsUnique(property, pkg))
        hasUniqueProperty = true;
    }

    if (!hasUniqueProperty) {
      PatternParameterValues vals =
        new PatternParameterValues(UniqueIdentifier.getPATTERN_ID());
      vals.addValue("context", klass);
      result .add(new AnalysisResult("Class␣"+klass.getName()+
                                    "␣does␣not␣have␣a␣unique␣key.",
              klass,
              UniqueIdentifier .getPATTERN_ID(),
              "UniqueIdentifier",
              vals ));
    }

    ...

}
```

The *instant fix* mechanism described in Section 7.2 is provided by action2 of class ClassModelAnalysisView. Upon invocation, this action looks up the currently highlighted item in the constraint elicitation view and instantiates the constraint patterns that are associated with the item. The following listing shows an excerpt of the makeActions() method in ClassModelAnalysisView.

```java
private void makeActions() {
action2 = new Action() {
  public void run() {
    // ask the table viewer which row is selected
    ClassModelAnalysisView view = (ClassModelAnalysisView) ViewActivationUtil.showView
                    ("com.ibm.bpia.rsa.modelanalysis.view.views.ClassModelAnalysisView");
    int index = view.getTable().getSelectionIndex();
    if (results!=null) {
      if (index<results.size()) {
        AnalysisResult res = results.get(index);
        Class contextClass = res.getContext();

        // instantiate patterns
        String[] patterns = res.getPatternId();
        if (patterns!=null && patterns.length>0) {
          final Package pkg = contextClass.getPackage();

          try {
            for (int i = 0; i < patterns.length; i++) {
              String id = patterns[i];
              final AbstractPatternInstance instance =
                PatternHelper.createPatternInstance(id, pkg, res.getPatternParameters());

              if (instance!=null) {
                // add instance to model and diagram
                final TransactionalEditingDomain editDomain =
                  TransactionUtil.getEditingDomain(pkg);
                final List<Diagram> diagrams = RSAHelper.getDiagramsForPackage(pkg);
                editDomain.getCommandStack().execute(
                  new RecordingCommand(editDomain){
                  protected void doExecute() {
                    for (Iterator iter = diagrams.iterator(); iter.hasNext();) {
                      Diagram diagram = (Diagram) iter.next();
                      Node node = new UMLDiagramHelper(editDomain).createNode(diagram,
                                                    (Element) instance.getBoundElement());
                      diagram.insertChild(node);
                    }
                  }
                });
                editDomain.getCommandStack().flush();
              }
            }
          } catch (RuntimeException e) { e.printStackTrace(); }
        }
      }
    }
    else showMessage("Action␣two␣executed␣on␣index␣"+index);
  }
};
action2.setText("Instant␣fix");
action2.setToolTipText("Instantiates␣appropriate␣constraint␣pattern(s)");
 ...
}
```

B.2 Constraint Specification

As explained, we implement our constraint pattern on top of RSA's pattern framework. In this pattern framework, each pattern is a subclass of AbstractPatternDefinition. We have extended this class by a general class ConstraintPattern, which itself has two subclasses, ElementaryPattern and CompositePattern. Subclasses of class ConstraintPattern must implement the following two abstract methods, on which we elaborate in the following two sections.

```
protected abstract Constraint transformToOCL(Collaboration collab);

public abstract IStatus isConsistent(Collaboration collab, IValidationContext ctx);
```

The class diagram in Figure B.2 gives an overview of our constraint pattern implementation.

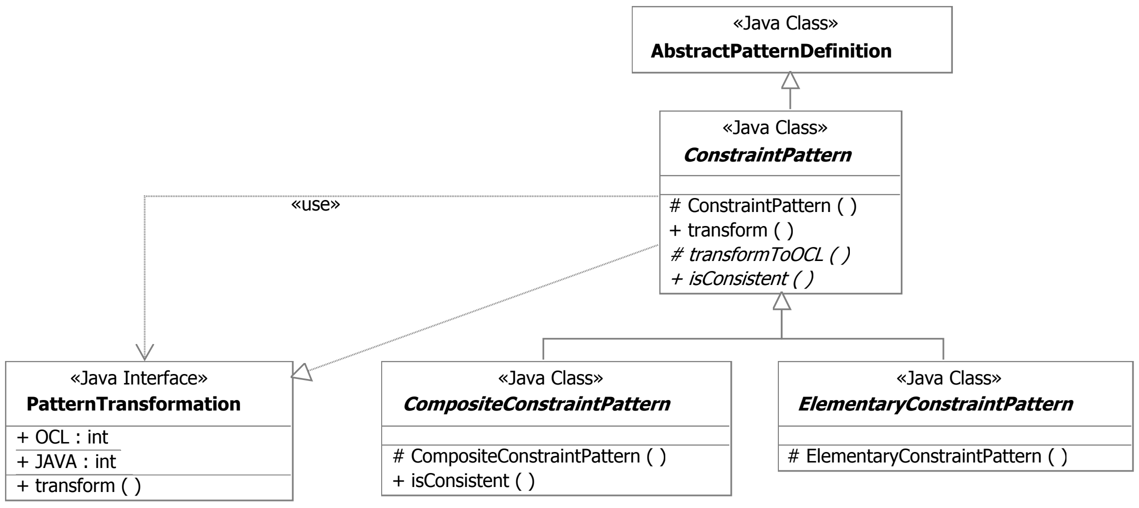


Figure B.2: Class diagram of constraint pattern API.

B.3 Consistency Analysis

As explained before, our consistency analysis builds on the validation framework in RSA. The validation framework distinguishes several validation outcomes: a model element can be successfully validated, or an informative message, a warning message, or an error message can be displayed. Since our analysis is a semi-decision approach, PatternInstanceConstraint returns a warning on unsuccessful validation.

Our analysis provides a new subclass of PatternInstanceConstraint that extends the class AbstractModelConstraint provided by the validation framework. The validate () method is invoked for each pattern instance in a given model and calls the isConsistent() method of the respective constraint pattern. Thus, the actual validation code, i. e., the checking of the consistency assumptions, is stored within each constraint pattern definition in the isConsistent() method. Figure B.3 shows a class diagram of this component.

The isConsistent() method analyzes pattern instances in a given validation context. The validation context, represented by an object of class IValidationContext, and the return type, IStatus, are defined in the validation framework of RSA. In the following, we provide

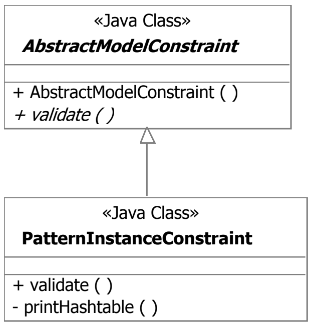


Figure B.3: Class diagram of the consistency analysis API.

an example implementation of the isConsistent() for the *Object In Collection* pattern that shows how to implement the method for a given constraint pattern.

```
public IStatus isConsistent(Collaboration collab, IValidationContext ctx) {
  List<org.eclipse.uml2.uml.Property> set =
    PatternHelper.getOneToManyProperties(collab, "set");
  Package pkg = collab.getNearestPackage();

  // checking consistency assumptions (i) + ( iii )
  for (org.eclipse.uml2.uml.Property p : set) {
    if (PatternHelper.propertyHasNoCycleRestriction(p, pkg))
      return ctx.createFailureStatus(new Object[]{collab.getName(),
                                                  PatternHelper.msgNoCycles(p)});
    else
    if (PatternHelper.propertyHasMultiplicityRestriction(p, pkg))
      return ctx.createFailureStatus(new Object[]{collab.getName(),
                                                  PatternHelper.msgMultRes(p)});
  }

  // checking consistency assumption (ii)
  org.eclipse.uml2.uml.Property pm = set.get(set.size()−1);
  if (!( pm.getUpper()>=1 || pm.getUpper()==−1))
    return ctx.createFailureStatus(new Object[]{collab.getName(),
       "The␣upper␣multiplicity␣bound␣of␣association␣end␣"+
        pm.getName()+"␣is␣smaller␣than␣1."});

  // consistency assumptions hold
  return ctx.createSuccessStatus();
}
```

B.4 Code Generation

Based on the transformation framework of RSA, COPACABANA defines a transformation that transforms pattern instances to OCL constraints. We integrate the transformation component of COPACABANA into the transformation framework by providing a new transformation rule CollaborationTransformation because in RSA, pattern instances are represented as UML collaborations. Figure B.4 shows a class diagram of the transformation component.

Similar to the consistency analysis component, the transformation rule invokes a special transformation method transformToOCL() that must be defined for each constraint pattern. This method transforms pattern instances. In particular, the return type of this

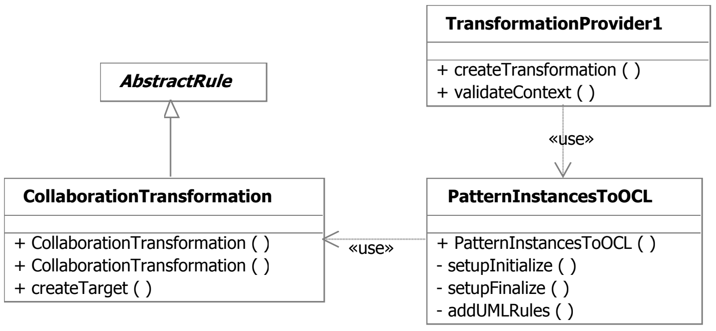


Figure B.4: Class diagram of COPACABANA's transformation component.

method is the UML class Constraint. The OCL expression is embedded in the body of the constraint. The following listing shows the implementation of this method for the *Object In Collection* pattern.

```
public Constraint transformToOCL(Collaboration collab) {
  List<org.eclipse.uml2.uml.Property> navigationList =
    PatternHelper.getOneToManyProperties(collab,"set");
  String navigation = UML2Helper.toOCLNavigation(navigationList);
  List<org.eclipse.uml2.uml.Property> elementList =
    PatternHelper.getOneToManyProperties(collab,"element");
  Class ownerClass = (Class) PatternHelper.getPatternInstanceParameter(collab, "context");

  if (navigation == null || ownerClass == null)
    return null;

  String element = "";
  if (elementList==null || elementList.size()==0)
    element = "self";
  else
    element = "self."+UML2Helper.toOCLNavigation(elementList);

  String body = "self." + navigation + "->includes("+element+")";
  return UML2Helper.createOCLConstraint(
      "Context␣object␣must␣be␣in␣" + navigation, body, ownerClass);
}
```

For instances of elementary patterns, the transformation consists of replacing formal by actual parameters in an OCL template as shown above. For instances of composite patterns, the transformation is recursively invoked on the composite pattern instances and their results are combined in another expression. The following listing shows how to transform composite constraint patterns by the example of the *Or* constraint pattern.

```
protected Constraint transformToOCL(Collaboration collab) {
  String body = "";
  List<Collaboration> operands =
    PatternHelper.getOneToManyCollaborations(collab, "operands");

  // iterate over the operands of this pattern instance and create body string
  Element context = null;
  for (Iterator iter = operands.iterator(); iter.hasNext();) {
    Collaboration operand = (Collaboration) iter.next();

    ConstraintPattern cpattern =
    (ConstraintPattern) PatternHelper.getPatternDefinitionFor(operand);
    Constraint op = cpattern.transform(operand,PatternTransformation.OCL);

    if (context == null)
      context = (Element) op.getConstrainedElements().get(0);
    String opBody = UML2Helper.getConstraintBody(op);
    body += opBody;
    if (iter.hasNext())
      body+=" or \n    ";
  }

  Constraint constraint = UML2Helper.createOCLConstraint("Or",
      body,
      context);

  return constraint;
}
```

Appendix C

Additional Constraints for the Monitor Model

In this appendix, we present additional constraints from our case study on the monitor model (Section 8.2). In particular, this appendix comprises constraints for situations, monitoring contexts, types, metrics, output slots, conditions, evaluation strategies, monitored entities, and cyclic dependencies. Furthermore, this section contains a special type of constraints: unsupported model elements.

C.1 Situations.

Constraint C.1.1 (triggered events 1). *Outgoing events triggered by situations can be populated by only one map.*

```
context SituationDefinition inv triggered_events_1:
  self.situationTriggeredEvent.mapDefinition->size() <= 1
```

```
context SituationDefinition inv triggered_events_1:
   MultiplicityRestriction ( SituationDefinition ,
                             situationTriggeredEvent.mapDefinition,
                             <=,
                             1)
```

Constraint C.1.2 (triggered events 2). *Situation can trigger outgoing events in their own contexts only.*

```
context SituationDefinition inv triggered_events_2:
  self.situationTriggeredEvent->forAll(e |
    e.monitoringContextDefinition = self.monitoringContextDefinition)
```

```
context SituationDefinition inv triggered_events_2:
  AttributeRelation ( SituationDefinition ,
                      monitoringContextDefinition,
                      =
                      situationTriggeredEvent,
                      monitoringContextDefinition)
```

Constraint C.1.3 (complex intervals). *The TimeIntervals for the on time based situation is restricted to reference a RecurringTimeIntervals that own a RecurrencePeriod only. That is the RecurringTimeIntervals does not own an AnchorPoint or TimeInterval.*

```
context SituationDefinition inv complex_intervals:
  self.evaluatedWhen.ownedEvaluationTime.recurringTimeIntervals
  −>forAll(i | i.anchorPoint.oclIsUndefined() and
               (i. interval .oclIsUndefined() or
                i. interval −>size()>0))
```

```
context SituationDefinition inv complex_intervals:
  ForAll( SituationDefinition ,
        evaluatedWhen.ownedEvaluationTime.recurringTimeIntervals,
        {LiteralOCL(RecurringTimeIntervals,"self.anchorPoint.oclIsUndefined()"),
         Or(LiteralOCL(RecurringTimeIntervals,"self. interval .oclIsUndefined ()"),
            MultiplicityRestriction (RecurringTimeIntervals,interval ,>,0))})
```

Figure C.1 summarizes the pattern instances on situations as represented in RSA. All three constraints in this section can be expressed using patterns.

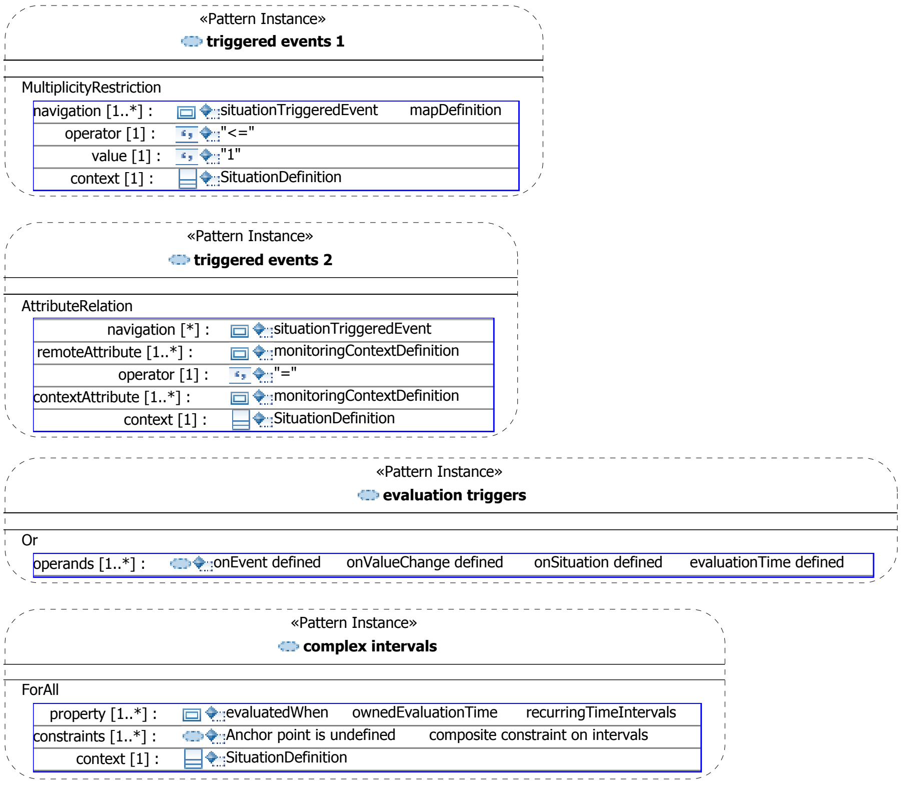

«Pattern Instance»
triggered events 1
MultiplicityRestriction
navigation [1..*] : situationTriggeredEvent mapDefinition
operator [1] : "<="
value [1] : "1"
context [1] : SituationDefinition
«Pattern Instance»
triggered events 2
AttributeRelation
navigation [*] : situationTriggeredEvent
remoteAttribute [1..*] : monitoringContextDefinition
operator [1] : "="
contextAttribute [1..*] : monitoringContextDefinition
context [1] : SituationDefinition
«Pattern Instance»
evaluation triggers
Or
operands [1..*] : onEvent defined onValueChange defined onSituation defined evaluationTime defined
«Pattern Instance»
complex intervals
ForAll
property [1..*] : evaluatedWhen ownedEvaluationTime recurringTimeIntervals
constraints [1..*] : Anchor point is undefined composite constraint on intervals
context [1] : SituationDefinition

Figure C.1: Pattern instances for situations.

C.2 Monitoring Contexts.

Constraint C.2.1 (context relations). *Two monitoring contexts can be related by at most one relation.*

```
context ParentContextRelationship inv context_relations:
not ParentContextRelationship.allInstances()
      −>exists(x, y | x<>y and x.childContextDefinition=y.childContextDefinition and
                                    x.parentContextDefinition=y.parentContextDefinition)
```

```
context ParentContextRelationship inv context_relations:
  UniquePath(MonitoringContextDefinition,childContextRelationship.childContextDefinition)
```

Constraint C.2.2 (context cycles). *The directed graph of monitoring context that are connected via context relations is acyclic.*

```
context MonitoringContextDefinition
def getChildren() = self.childContextRelationship.childContextDefinition−>union(
       self.childContextRelationship.childContextDefinition.allChildContexts())
def getAncestors() = self.parentContextRelationship.parentContextDefinition−>union(
       self.parentContextRelationship.parentContextDefinition.getAncestors())
def getRelatedContexts() = getChildren()−>union(getAncestors())

inv context_cycles: self.getChildren()−>excludes(self)
```

```
context MonitoringContextDefinition inv context_cycles:
  NoCyclicDependency(MonitoringContextDefinition,
                     childContextRelationship.childContextDefinition)
```

Constraint C.2.3 (member identification). *All counters, timers, and keys in the context should have unique names.*

```
context MonitoringContextDefinition inv member_identification:
  self.counterDefinition−>union(self.timerDefinition−>union(self.keyDefinition))
    −>isUnique(name)
```

This constraint cannot be expressed using any of our patterns. Although the *Unique Identifier* pattern appears to fit, it is defined for uniqueness of a set of properties of the *same* class. Constraint *member identification* requires the uniqueness of one property shared by *several* classes. This constraint may be a candidate for a new constraint pattern. In Section 8.2.4, we discuss the elicitation of new constraint patterns from constraints that cannot (yet) be expressed using our pattern library.

Constraint C.2.4 (terminating situations). *A terminating situation trigger of a monitoring context must be defined in the same context.*

```
context MonitoringContextDefinition inv terminating_situations :
  self.terminatedBy−>forAll(s | s.monitoringContextDefinition = self)
```

```
context MonitoringContextDefinition inv terminating_situations :
  AttributeRelation(MonitoringContextDefinition,terminatedBy,=,monitoringContextDefinition)
```

Constraint C.2.5 (context reference). *Elements in a context should only refer to elements in another context if they exist within the same relationship path. It is not supported for any monitor-model element to refer to another monitoring context that is not one of its ancestors, a direct descendant.*

This constraint ensures that the parameters for map calculations stay within valid bounds of monitoring context definitions.

```
context MonitoringContextDefinition inv context_reference:
  self. slotDefinitions .mapDefinition−>
      forAll (map | map.checkForAggregateFunction())

context MapDefinition
def: checkForAggregateFunction() : Boolean =
      self.outputValue.value−>forAll(value |
        if  (value.oclIsTypeOf(StructuredOpaqueExpression))
        then let exp = value.oclAsType(StructuredOpaqueExpression).expression in
        expressionAccessesOnlyAncestorOrSame(exp, map.monitoringContextDefinition) and
          (expressionAccessesOnlyChild(exp, map.monitoringContextDefinition) implies
          checkMPEisUnderAggregate(exp))
        else true
        endif)

def: expressionAccessesOnlyAncestorOrSame(exp: Expression exp,
                                   mcd:MonitoringContextDefinition) : Boolean =
      if  (exp.oclIsTypeOf(UnaryOperatorExpression))
      then expressionAccessOnlyAncestorOrSame(exp.expression, mcd)
      else
        if  (exp.oclIsTypeOf(BinaryOperatorExpression))
        then expressionAccessOnlyAncestorOrSame(exp.firstOperand, mcd) and
              expressionAccessOnlyAncestorOrSame(exp.secondOperand, mcd)
        else
          if  (exp.oclIsTypeOf(FunctionExpression))
          then exp.oclAsType(FunctionExpression).arguments.argumentValue−>
                forAll (exp |  expressionAccessesOnlyAncestorOrSame(exp,mcd))
          else
            if  (exp.oclIsTypeOf(ModelPathExpression))
            then exp.oclAsType(ModelPathExpression).steps−>select(s |
                  s.oclIsTypeOf(ReferenceStep)).referencedObject−>forAll(o |
                  o.monitoringContextDefinition.getAncestors()−>includes(mcd))
            else true
            endif
          endif
        endif
      endif

def: expressionAccessesOnlyChild(exp: Expression exp,
                                      mcd:MonitoringContextDefinition) : Boolean =
      if  (exp.oclIsTypeOf(UnaryOperatorExpression))
      then expressionAccessOnlyChild(exp.expression, mcd)
      else
        if  (exp.oclIsTypeOf(BinaryOperatorExpression))
        then expressionAccessOnlyChild(exp.firstOperand, mcd) and
              expressionAccessOnlyChild(exp.secondOperand, mcd)
        else
          if  (exp.oclIsTypeOf(FunctionExpression))
          then exp.oclAsType(FunctionExpression).arguments.argumentValue−>
                forAll (exp |  expressionAccessesOnlyChild(exp,mcd))
          else
            if  (exp.oclIsTypeOf(ModelPathExpression))
            then exp.oclAsType(ModelPathExpression).steps−>select(s |
            s.oclIsTypeOf(ReferenceStep)).referencedObject->forAll(o |
            o.monitoringContextDefinition.getChildren()->includes(mcd))
       else true
       endif
      endif
     endif
    endif

def: checkMPEisUnderAggregate(exp: Expression exp) : Boolean =
     if  (exp.oclIsTypeOf(UnaryOperatorExpression))
    then checkMPEisUnderAggregate(exp.expression, mcd)
    else
     if  (exp.oclIsTypeOf(BinaryOperatorExpression))
     then checkMPEisUnderAggregate(exp.firstOperand, mcd) and
          checkMPEisUnderAggregate(exp.secondOperand, mcd)
     else
      if  (exp.oclIsTypeOf(FunctionExpression))
      then exp.oclAsType(FunctionExpression).arguments.argumentValue->
             forAll (exp2 |
               if  (exp2.oclIsTypeOf(ModelPathExpression))
               then exp.oclAsType(FunctionExpression).isAggregate()
               else checkMPEisUnderAggregate(exp2))
               endif
      else true
      endif
     endif
    endif

context FunctionDefinition
def: isAggregate() : Boolean =
    Set{'Sum', 'Avg', 'StdDev', 'SumProd', 'Count',
        'MaxV', 'MinV', 'Every', 'Exist'}->includes(self.definition.functionName)
```

Since such complex, domain-specific constraints comprise numerous method definitions, they cannot be covered by our patterns.

Constraint C.2.6 (key maps). *All ParentContextRelationships should reference all parent monitoring contexts KeyDefinitions through parent Key Maps.*

```
context MonitoringContextDefinition inv key_maps:
  self.keyDefinition->forAll(x | self.childContextRelationShip.parentKey.outputSlot->
                                  forAll (y | x = y))
```

Unfortunately, the *ForAll* pattern does not match this constraint. Therefore, it cannot be expressed using our pattern library.

Constraint C.2.7 (context auto creation). *If the auto creation of parent monitoring contexts is done more than one level up in the relation hierarchy, we assume that starting from the second level up, the parent key maps will refer to previous key definitions only, because these are the only metrics that will contain values in this case. On the other hand the first level parent could access the child key definitions metrics as well as any child metrics deducible from the event entry fields.*

```
context MonitoringContextDefinition
def: keyReference(r:ParentContextRelationship):Boolean =
      r.parentContextDefinition.parentContextRelationship->forAll(pr | keyReference(pr)) and
      r.parentKey.input->forAll(i |
          if  i.oclIsTypeOf(MetricDefinition)
          then r.childContextDefinition.inboundEventDefinition->exists(m |
            isDeducible(i.oclAsType(MetricDefinition),
                        m.oclAsType(MetricDefinition)))
          else i.oclIsTypeOf(KeyDefinition)
          endif)

def: isDeducible(m1:MetricDefinition,m2:MetricDefinition) =
      if (not m1.oclIsTypeOf(ReadWriteMetricDefinition))
      then false
      else m1.mapDefinition->exists(map |
        (map.input->size() = 1 and map.input.any() = m2) or
        (map.input->size() > 1 and map.input->forAll(m3 | isDeducible(m3,m2))))

inv context_auto_creation:
  self.parentContextRelationship->forAll(r1 | r1.parentContextAutoCreated implies
     r.parentContextRelationship->forAll( r2 | keyReference(r2)))
```

This constraint defines two complex functions, keyReference() and isDeducible. Thus, we cannot express it using our pattern library.

Constraint C.2.8 (no parent map). *A parent metric can have its map in the child however the opposite can not occur.*

```
context MapDefinition
inv no_parent_map:
  self.input->forAll( i | self.getContext() = i.getContext() or
                          self.getContext().getAncestors()->includes(i.getContext()))
```

Since this constraint employs numerous user-defined functions, we cannot express it using our predefined constraint patterns.

Constraint C.2.9 (context id). *All monitoring contexts in the monitor model should have unique names.*

```
context MonitoringContextDefinition inv context_id:
  MonitoringContextDefinition:allInstances()->isUnique(name)
```

```
context MonitoringContextDefinition inv context_id:
        UniqueIdentifier(MonitoringContextDefinition, name)
```

Constraint C.2.10 (relation id). *All Relations and their maps should have unique names.*

```
context ParentContextRelationship inv relation_id:
     ParentContextRelationship::allInstances()->isUnique(name) and
     ParentContextRelationship::allInstances().parentKey->isUnique(name)
```

When representing this constraint using patterns, we split it into two constraints. This is possible because in UML, all constraints of the same element are implicitly conjunct [Object Management Group (OMG), 2006c].

```
context ParentContextRelationship inv relation_id_1 :
        UniqueIdentifier (ParentContextRelationship,name)

context ParentContextRelationship inv relation_id_1 :
        UniqueIdentifier (ParentContextRelationship,parentKey.name)
```

Constraint C.2.11 (key ownership). *The foreignKeyDefinition of a ParentContextRelationship must be owned by the childContextDefinition; the parentKeyDefinition must be owned by the parentContextDefinition.*

```
context ParentContextRelationship inv key_ownership:
  parentKey−>forAll( m |
    m.input−>size() = 1 and
    m.outputSlot−>size() = 1 and
    self. childContextDefinition . slotDefinition −>includes( m.input−>at(1) ) and
    self.parentContextDefinition. slotDefinition −>includes( m.outputSlot ) )
```

```
context ParentContextRelationship inv key_ownership:
  ForAll (ParentContextRelationship,
         parentKey,
         {  MultiplicityRestriction  (MapDefinition,outputSlot ,=,1),
          LiteralOCL(MapDefinition,"parentContextRelationship.childContextDefinition
                                        −>includes(self.input−>at(1))"),
          LiteralOCL(MapDefinition,"parentContextRelationship. slotDefinition
                                       −>includes(self.outputSlot ))"),
            MultiplicityRestriction  (MapDefinition,input,=,1)}
         )
```

Constraint C.2.12 (parent keys). *Each KeyDefinition of the parentContextDefinition must be a parentKeyDefinition of a ParentContextRelationship.*

```
context ParentContextRelationship inv parent_keys:
  parentContextDefinition. keyDefinition −>forAll( k | parentKey.outputSlot−>includes( k ) )
```

```
context ParentContextRelationship inv parent_keys:
  ForAll (ParentContextRelationship,
         parentContextDefinition,
         {LiteralOCL(KeyDefinition,"monitoringContextDefinition.parentKey.outputSlot
                                       −>includes(self)")}
        )
```

Constraint C.2.13 (foreign key dependency). *A foreignKeyDefinition of a ParentContextRelationship may only depend on input slots in its own monitoring context.*

```
context ParentContextRelationship inv foreign_key_dependency:
parentKey−>forAll( m |
   if  m.input−>at(1).oclIsKindOf( OutputSlotDefinition )
  then m.input.upstreamSlotDefinitions()−>forAll( s |
        self. childContextDefinition . slotDefinition −>includes( s ))
  endif  )
```

```
context OutputSlotDefinition
def: upstreamSlotDefinitions() : Set( InputSlotDefinition ) =
  let parents = self.mapDefinition.input in
  parents−>union(parents−>select(i |
     i.oclIsTypeOf(OutputslotDefinition )). upstreamSlotDefinitions())
```

This constraint cannot be represented using our constraint patterns because it involves the definition of a complex function.

Constraint C.2.14 (context ancestors). *The contexts of the metrics connected by a RelatedMetric need to have a closest common ancestor context, or no common ancestor context at all.*

```
context RelatedMetric inv context_ancestors:
  let mcd1 = self.KPIDefinition.monitoringContextDefinition in
  let mcd2 = self.metric.monitoringContextDefinition in
    mcd1.hasClosestCommonAncestorWith(mcd2) ||
    m1.getAncestors()−>intersect(mcd2.getAncestors())−>size() = 0


context MonitoringContextDefinition
def: hasClosestCommonAncestorWith(b2: MonitoringContextDefinition): Boolean =
   self.getAncestors()−>intersect(b2.getAncestors())−>
     select( ancestor | isStraightAncestorOf(self,ancestor) and
                        isStraightAncestorOf(b2,ancestor))−>
      forAll (c | self.getAncestors()−>intersect(b2.getAncestors())−>
      forAll (e | isAncestorOf(c, e)))

def: isAncestorOf(mcd: MonitoringContextDefinition) : Boolean =
      if (self=mcd)
     then false
     else mcd.getAncestors()−>includes(self)
     endif

def: isStraightAncestorOf(mcd: MonitoringContextDefinition) : Boolean =
      if (self=mcd)
     then true
     else mcd.getAncestors()−>count(mcd) = 1
     endif
```

This constraint cannot be represented using our constraint patterns because it involves the definition of complex functions.

In Figure C.2, we show the pattern instances on monitoring contexts as represented in RSA. Out of 14 constraints in this section, 7 can be expressed using patterns.

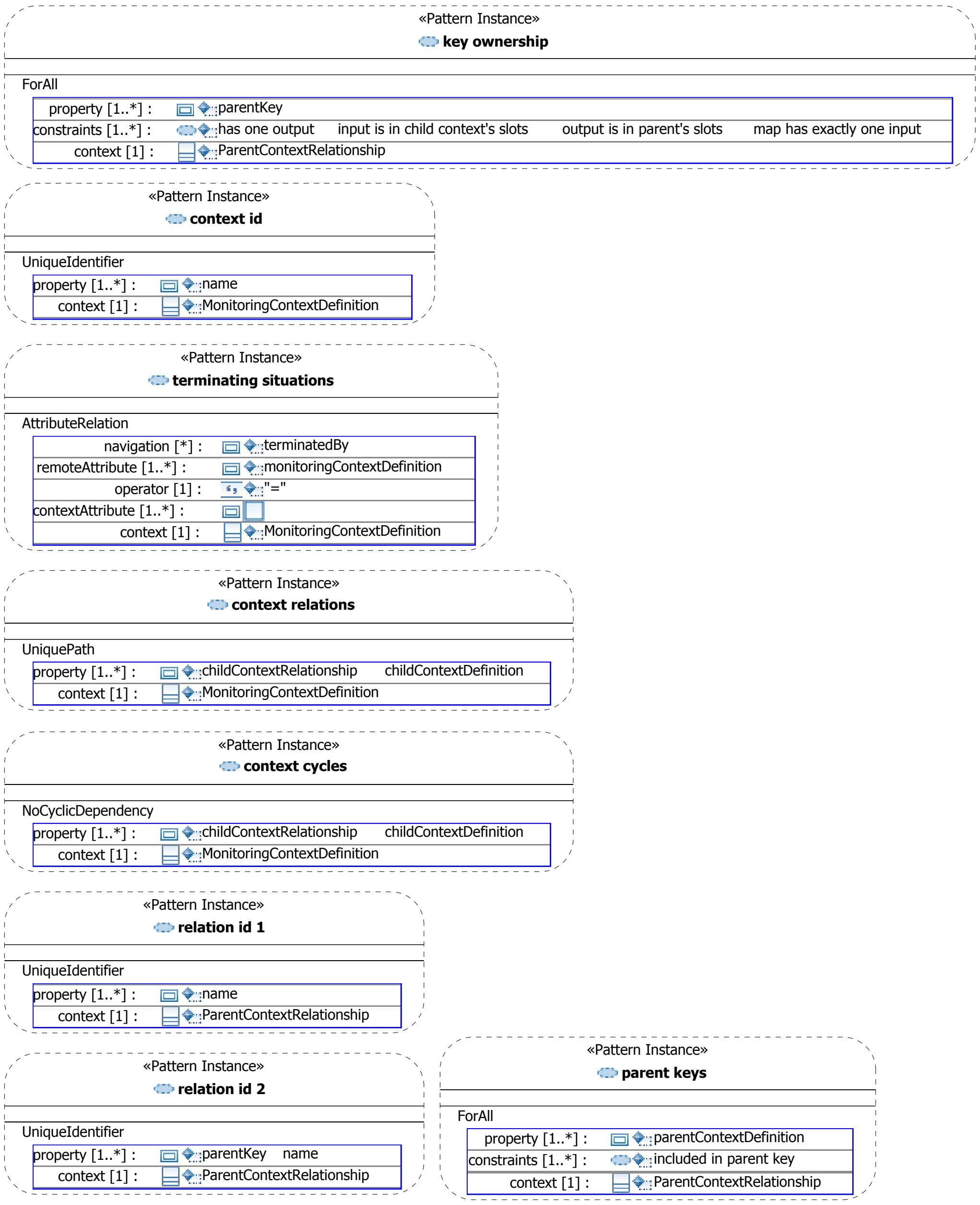

«Pattern Instance»
key ownership
ForAll
property [1..*] : parentKey
constraints [1..*] : has one output input is in child context's slots output is in parent's slots map has exactly one input
context [1] : ParentContextRelationship
«Pattern Instance»
context id
UniqueIdentifier
property [1..*] : name
context [1] : MonitoringContextDefinition
«Pattern Instance»
terminating situations
AttributeRelation
navigation [*] : terminatedBy
remoteAttribute [1..*] : monitoringContextDefinition
operator [1] : "="
contextAttribute [1..*] :
context [1] : MonitoringContextDefinition
«Pattern Instance»
context relations
UniquePath
property [1..*] : childContextRelationship childContextDefinition
context [1] : MonitoringContextDefinition
«Pattern Instance»
context cycles
NoCyclicDependency
property [1..*] : childContextRelationship childContextDefinition
context [1] : MonitoringContextDefinition
«Pattern Instance»
relation id 1
UniqueIdentifier
property [1..*] : name
context [1] : ParentContextRelationship
«Pattern Instance»
relation id 2
UniqueIdentifier
property [1..*] : parentKey name
context [1] : ParentContextRelationship
«Pattern Instance»
parent keys
ForAll
property [1..*] : parentContextDefinition
constraints [1..*] : included in parent key
context [1] : ParentContextRelationship

Figure C.2: Pattern instances for monitoring contexts and their relations.

C.3 Typed Elements / Types.

Constraint C.3.1 (typed elements). *All typed elements in the model should have types, except for timers and counters.*

```
context TypedElement inv typed_elements:
(not (self.oclIsKindOf(TimerDefinition) or
  self.oclIsKindOf(CounterDefinition)))
  implies not self.type.oclIsUndefined()
```

```
context TypedElement inv typed_elements:
  IfThenElse(LiteralOCL(TypedElement,"not (self.oclIsKindOf(TimerDefinition) or
                                          self.oclIsKindOf(CounterDefinition )")),
             MultiplicityRestriction (TypedElement,type,=,1),
            )
```

Constraint C.3.2 (primitive types). *Primitive types should map to one of the supported types. The set of supported types is { Integer, Boolean, String, UnlimitedNatural, Duration, Time, Real }.*

```
context TypedElement inv primitive_types:
 self.type.oclIsKindOf(PrimitiveType) implies
 (self.type.oclIsTypeOf(Integer) or
  self.type.oclIsTypeOf(Boolean) or
  self.type.oclIsTypeOf(String) or
  self.type.oclIsTypeOf(UnlimitedNatural) or
  self.type.oclIsTypeOf(Duration) or
  self.type.oclIsTypeOf(Time) or
  self.type.oclIsTypeOf(Real)
 )
```

```
context TypedElement inv primitive_types:
  IfThenElse(TypeRelation(TypedElement,type,PrimitiveType),
            TypeRestriction(TypedElement,type,{Integer,
                                               Boolean,
                                               String,
                                               UnlimitedNatural,
                                               Duration,
                                               Time,
                                               Real}),
            )
```

Constraint C.3.3 (metric types). *Metric types and structured metric fields cannot be abstract.*

```
context DataType inv metric_types:
DataType.allInstances()->
  union(EventType.allInstances)->forAll( x |
    not x.ownedAttribute.type.oclAsType(Classifier). isAbstract)
```

```
context DataType inv metric_types:
  AttributeValueRestriction (DataType,ownedAttribute.type.isAbstract,=,false)
```

Constraint C.3.4 (event superclassifier). *EventType must have at most one superClassifier, which must be another EventType.*

```
context EventType inv event_superclassifier:
  self.superClassifier->size() = 1 and
  self.superClassifier->forAll( c | c.oclIsTypeOf(EventType) )
```

```
context EventType inv event_superclassifier_1:
   MultiplicityRestriction (EventType,superClassifier,=,1)

context EventType inv event_superclassifier_2:
  TypeRestriction(EventType,superClassifier,{EventType})
```

Constraint C.3.5 (composite aggregation). *The aggregation of event properties must be composite for EventType.*

```
context EventType inv composite_aggregation:
  self.ownedAttribute->forAll( p | p.aggregation = AggregationKind::composite )
```

```
context EventType inv composite_aggregation:
  ForAll (EventType,
        ownedAttribute,
        { AttributeValueRestriction (Property,aggregation,=,AggregationKind::composite)})
```

Constraint C.3.6 (dynamic properties). *Event properties must not be static for EventType.*

```
context EventType inv dynamic_properties:
  self.ownedAttribute->forAll( p | not p. isStatic  )
```

```
context EventType inv dynamic_properties:
  ForAll (EventType,
        ownedAttribute,
        { AttributeValueRestriction (Property, isStatic ,=, false )})
```

Constraint C.3.7 (defaultvalue type). *The type of the defaultValues of a metric must conform to its type.*

```
context MetricDefinition  inv  defaultvalue_type :
defaultValue.type->union(defaultValue.type.oclAsType(Classifier).allSuperClassifiers ())
  ->includes(type)

context Classifier
def:  allSuperClassifiers ()  :  Set( Classifier )  =
   self . superClassifier->union(self.superClassifier.allSuperClassifiers ())
```

This constraint cannot be represented using our constraint patterns because it involves calling a user-defined operation.

Constraint C.3.8 (history type). *The type of the target MetricDefinition of a historyRange association must be Duration or UnlimitedNatural of a metric.*

```
context MetricDefinition inv history_type :
 keepHistoryFor.type = Duration or
 keepHistoryFor.type = UnlimitedNatural
```

```
context MetricDefinition inv history_type :
  TypeRestriction( MetricDefinition ,
                   keepHistoryFor,
                   {Duration, UnlimitedNatural})
```

Constraint C.3.9 (timer type). *The type of a TimerDefinition must be Duration.*

```
context TimerDefinition inv timer_type:
  self.type.oclIsKindOf(Duration)
```

```
context TimerDefinition inv timer_type:
  TypeRestriction(CounterDefinition,type,{Duration})
```

Constraint C.3.10 (counter type). *The type of a CounterDefinition must be an integer type.*

```
context CounterDefinition inv counter_type:
  self.type.oclIsKindOf(Integer) or
  self.type.oclIsKindOf(UnlimitedNatural)
```

```
context CounterDefinition inv counter_type:
  TypeRestriction(CounterDefinition,type,{Integer,UnlimitedNatural})
```

Constraint C.3.11 (slot type). *SlotDefinitions must reference a Type.*

```
context SlotDefinition inv slot_type :
 self.type->notEmpty()
```

```
context SlotDefinition inv slot_type :
   MultiplicityRestriction ( SlotDefinition ,type,>,0)
```

In Figure C.3, we show the pattern instances on typed elements as represented in RSA. Out of 11 constraints in this section, 10 can be expressed using patterns.

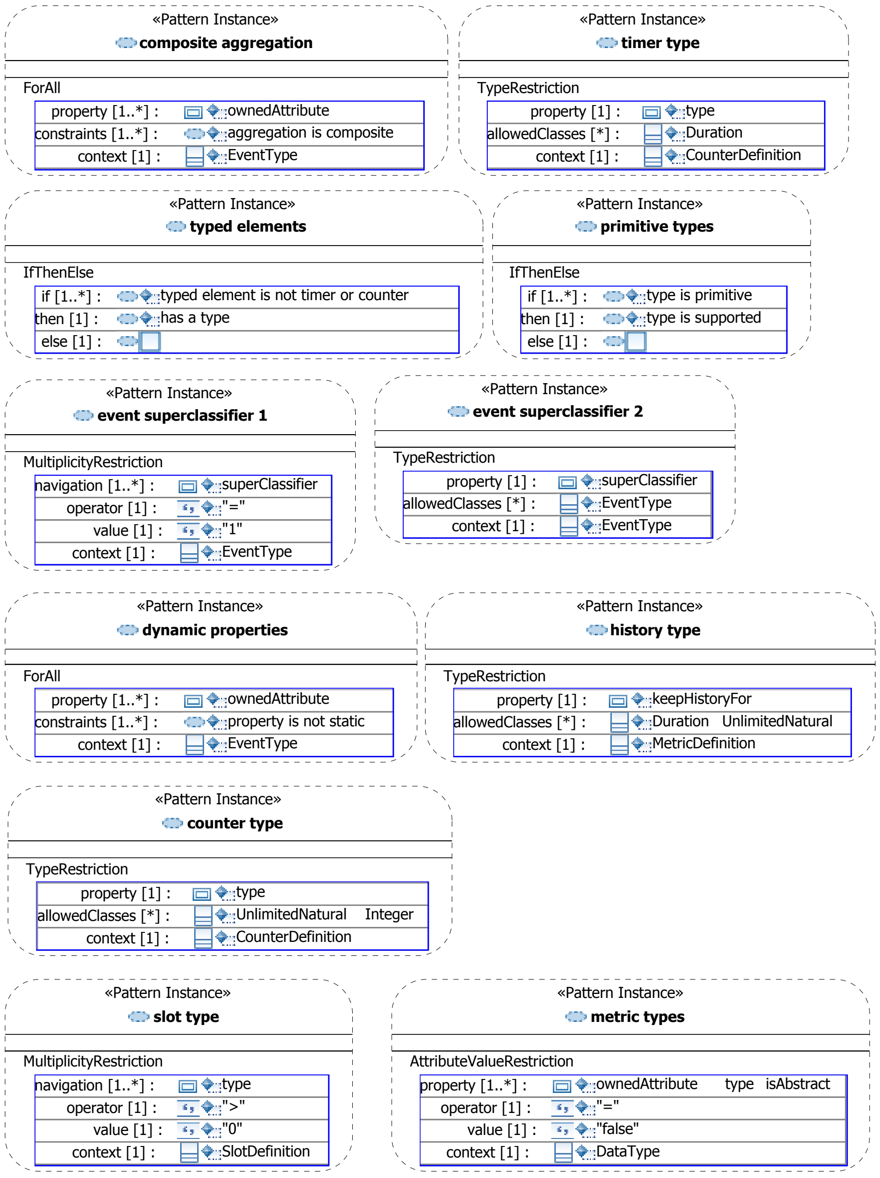


Figure C.3: Pattern instances for typed elements.

C.4 Metrics, Counters, Timers.

Constraint C.4.1 (max default values). *The number of defaultValues must not exceed the upper multiplicity bound of a metric.*

```
context MetricDefinition inv max_default_values:
  defaultValue−>size() <= upperBound.value
```

```
context MetricDefinition inv max_default_values:
   MultiplicityRestriction (MetricDefinition ,defaultValue,<=,upperBound.value)
```

Constraint C.4.2 (history size). *For each metric, the upper multiplicity bound of the target MetricDefinition of a keepHistoryFor association must be one.*

```
context MetricDefinition inv history_size :
  keepHistoryFor.upperBound.value = 1
```

```
context MetricDefinition inv history_size :
  AttributeValueRestriction ( MetricDefinition ,keepHistoryFor.upperBound.value,=,1)
```

Constraint C.4.3 (history range context). *The target MetricDefinition of a historyRange association must be owned by the same or a parent MonitoringContextDefinition of that of a metric.*

```
context MetricDefinition inv history_range_context:
Set{monitoringContextDefinition}->
          union(monitoringContextDefinition.parentContextRelationship.
                parentContextDefinition->asSet())
          ->includes(keepHistoryFor.monitoringContextDefinition)
```

In the pattern representation, we replace the original set operation by an instance of the *Or* pattern. This is possible because of the equivalence $x \in \bigcup_i A_i \equiv \bigvee_i x \in A_i$.

```
context MetricDefinition inv history_range_context:
  Or(ObjectInCollection( MetricDefinition ,
                          keepHistoryFor.monitoringContextDefinition,
                          monitoringContextDefinition),
     ObjectInCollection( MetricDefinition ,
                keepHistoryFor.monitoringContextDefinition,
                 monitoringContextDefinition.parentContextRelationship.parentContextDefinition))
```

Since such conversion requires significant user sophistication, it should be considered to introduce a new constraint pattern that helps to express constraint *history range context* in a more concise way.

Constraint C.4.4 (output size). *The number of values in metrics must not exceed the multiplicity of the output value.*

```
context ReadWriteMetricDefinition inv output_size:
  mapDefinition.outputValue->size()<= upperBound.value
```

```
context ReadWriteMetricDefinition inv output_size:
   MultiplicityRestriction  (ReadWriteMetricDefinition,mapDefinition.outputValue,
                             <=,upperBound.value)
```

Constraint C.4.5 (counter context). *The incrementedWhen, decrementedWhen, and setToZeroWhen SituationDefinitions must be owned by the same MonitoringContextDefinition that owns the CounterDefinition.*

```
context CounterDefinition inv counter_context:
  let  situations  = self.decrementedWhen->
                    union(self.setToZeroWhen->union(self.incrementedWhen)) in
  situations ->forall(s | s.monitoringContextDefinition = self.monitoringContextDefinition)
```

We split this constraint into three parts in order to represent it with constraint patterns. This is possible because of the equivalence $(\forall s \in \bigcup_i A_i).P(s) \equiv \bigwedge_i (\forall s \in A_i).P(s)$.

```
context CounterDefinition inv counter_context_1:
  AttributeRelation (CounterDefinition,
                     monitoringContextDefinition,
                     =,
                     decrementedWhen,
                     monitoringContextDefinition)

context CounterDefinition inv counter_context_2:
  AttributeRelation (CounterDefinition,
                     monitoringContextDefinition,
                     =,
                     incrementedWhen,
                     monitoringContextDefinition)

context CounterDefinition inv counter_context_3:
  AttributeRelation (CounterDefinition,
                     monitoringContextDefinition,
                     =,
                     setToZeroWhen,
                     monitoringContextDefinition)
```

Constraint C.4.6 (timer context). *The startedWhen, stoppedWhen, and resetWhen SituationDefinitions must be owned by the same MonitoringContextDefinition that owns the TimerDefinition, or by one with which it has a MonitoringContextRelationship.*

```
context TimerDefinition inv timer_context:
  let situations = self.startedWhen->union(self.stoppedWhen)->union(self.resetWhen) in
  situations->forAll( s | s.monitoringContextDefinition = self.monitoringContextDefinition )
```

We split this constraint into three parts in order to represent it with constraint patterns.

```
context TimerDefinition inv timer_context_1:
  AttributeRelation (TimerDefinition,
                     monitoringContextDefinition,
                     =,
                     startedWhen,
                     monitoringContextDefinition)

context TimerDefinition inv timer_context_2:
  AttributeRelation (TimerDefinition,
                     monitoringContextDefinition,
                     =,
                     stoppedWhen,
                     monitoringContextDefinition)

context TimerDefinition inv timer_context_3:
  AttributeRelation (TimerDefinition,
                     monitoringContextDefinition,
                     =,
                     resetWhen,
                     monitoringContextDefinition)
```

In Figure C.4, we show the pattern instances on metrics as represented in RSA. All 6 constraints in this section can be expressed using patterns.

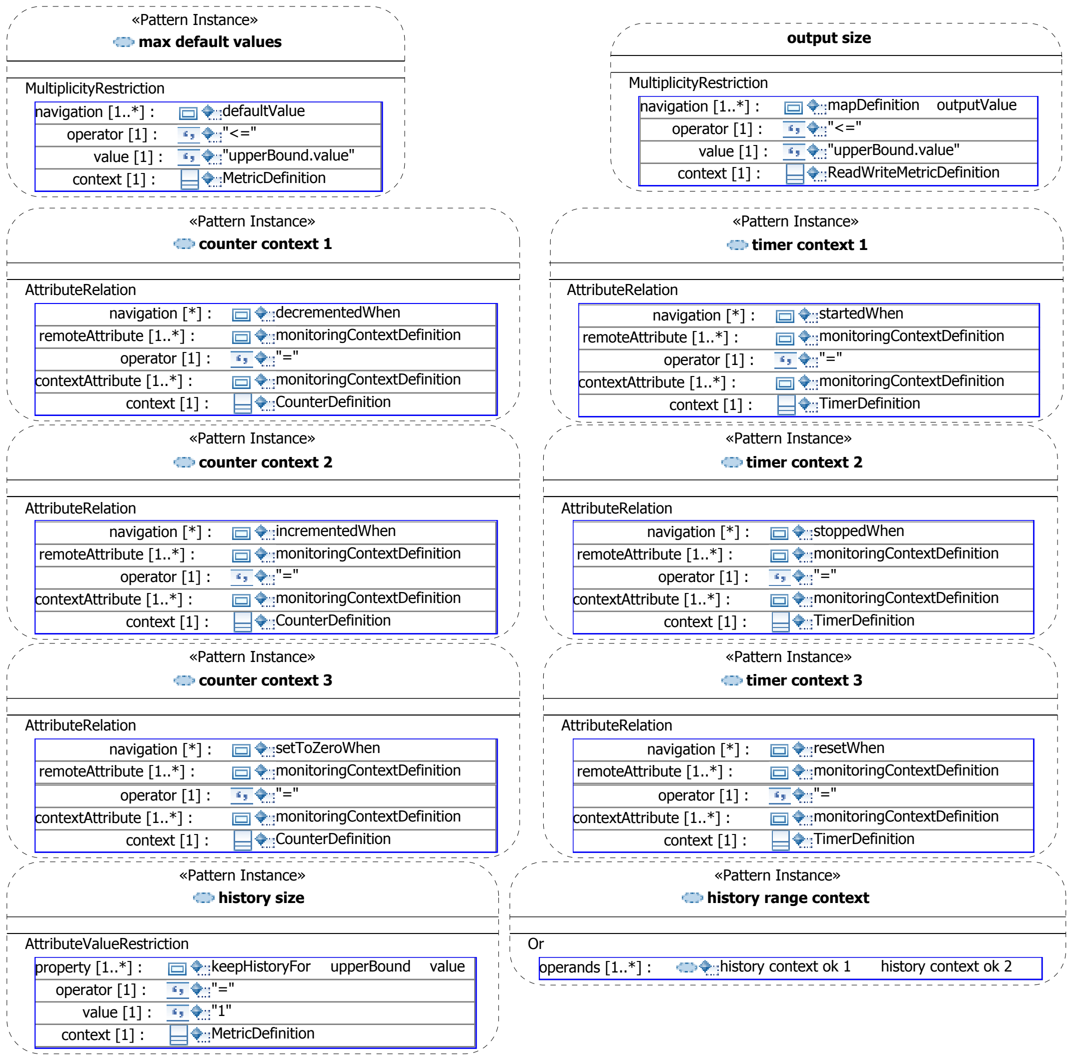

«Pattern Instance»
max default values
MultiplicityRestriction
navigation [1..*] : ::defaultValue
operator [1] : ::"<="
value [1] : ::"upperBound.value"
context [1] : ::MetricDefinition
output size
MultiplicityRestriction
navigation [1..*] : ::mapDefinition outputValue
operator [1] : ::"<="
value [1] : ::"upperBound.value"
context [1] : ::ReadWriteMetricDefinition
«Pattern Instance»
counter context 1
AttributeRelation
navigation [*] : ::decrementedWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::CounterDefinition
«Pattern Instance»
timer context 1
AttributeRelation
navigation [*] : ::startedWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::TimerDefinition
«Pattern Instance»
counter context 2
AttributeRelation
navigation [*] : ::incrementedWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::CounterDefinition
«Pattern Instance»
timer context 2
AttributeRelation
navigation [*] : ::stoppedWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::TimerDefinition
«Pattern Instance»
counter context 3
AttributeRelation
navigation [*] : ::setToZeroWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::CounterDefinition
«Pattern Instance»
timer context 3
AttributeRelation
navigation [*] : ::resetWhen
remoteAttribute [1..*] : ::monitoringContextDefinition
operator [1] : ::"="
contextAttribute [1..*] : ::monitoringContextDefinition
context [1] : ::TimerDefinition
«Pattern Instance»
history size
AttributeValueRestriction
property [1..*] : ::keepHistoryFor upperBound value
operator [1] : ::"="
value [1] : ::"1"
context [1] : ::MetricDefinition
«Pattern Instance»
history range context
Or
operands [1..*] : ::history context ok 1 history context ok 2

Figure C.4: Pattern instances for metrics, counters, and timers.

C.5 Output Slots.

Constraint C.5.1 (value structure). *It is assumed that the hierarchical structure of the ValueSpecification owned by the map coincides with the structure of the output slot of this map.*

```
context OutputSlotDefinition
def validateSlotStructure (vSpecList: Set(ValueSpecification), dataType:Type) : Boolean =
  vSpecList−>forAll(vSpec |
     if  (vspec.oclIsTypeOf(InstanceValue))
    then vspec.oclAsType(InstanceValue).ownedInstance.slot−>forAll(slot |
       if  ( slot .definingFeature.oclIsUndefined())
      then false
      else   if  ( slot .definingFeature.type.oclIsKindOf(DataType))
             then  slot .definingFeature.type.oclAsType(DataType).ownedAttribute−>
                  exists (pty  |  pty .type  =  slot .definingFeature.type  and
                               validateSlotStructure ( slot . value,  slot .definingFeature.type))
             else  if  ( slot .definingFeature.type.oclIsKindOf(PrimitiveType))
                  then dataType.oclAsType(DataType).ownedAttribute−>
                      exists (pty  |  pty .getType = slot .definingFeature.type)
```

```
                    else true
                    endif
               endif
        endif)
      else true
      endif)

inv value_structure:
  if self.type.oclIsTypeOf(EventType)
  then false
  else validateSlotStructure(self.mapDefinition.outputValue,self.type)
  endif
```

Since this constraint employs a user-defined function, we cannot express it using our predefined constraint patterns.

Constraint C.5.2 (targetslot type). *The type of an OutputSlotValueSpecification must conform to the type of the targetSlotDefinition.*

```
context OutputSlotValueSpecification inv targetslot_type:
  self.value->forAll( v | v.type->union( v.type.allSuperClassifiers )->
    includes( targetOutputSlotDefinition .type))
```

This constraint cannot be represented using our patterns because it uses set operators nested in a quantifier.

C.6 Conditions.

Constraint C.6.1 (condition parameter). *Only the inputs of a condition may parameterize its expression.*

```
context Condition inv condition_parameter:
  self.input.value.getInputs()->forAll( i | self.input->includes(i))

context OpaqueExpression
def: getInputs() : Set( InputSlotDefinition ) =
       self.contents->select(o | o.oclIsTypeOf(ReferenceStep)).referencedObject->
       select(o | o.oclIsTypeOf( InputSlotDefinition ))
```

Since this constraint employs a user-defined function, we cannot express it using our constraint patterns.

Constraint C.6.2. *Each parameter of a condition must be owned by a MonitoringContextDefinition that is related to the one hosting the ParameterizedExpression.*

```
context Condition inv:
parameter.monitoringContextDefinition->forAll( m |
          m.getRelatedContexts().slotDefinition->select(
          oclIsTypeOf( InboundEventDefinition ) )->collect(correlationPredicate)->union(
          m.getRelatedContexts().slotDefinition->select(
          oclIsTypeOf( InboundEventDefinition ) )->collect(  filter  ) )->union(
          m.getRelatedContexts().slotDefinition->select(
          oclIsTypeOf( OutboundEventDefinition ) )->collect(  filter  ) )->union(
          m.getRelatedContexts().situationDefinition−>collect(
          gatingCondition ) )−>union(
          m.getRelatedContexts().slotDefinition−>select(
          oclIsKindOf( OutputSlotDefinition ) )−>collect(
          mapDefinition.outputValueSpecification ) )−>includes( self ) )
```

Since this constraint uses complicated filtering on sets, we cannot express it using our constraint patterns.

C.7 Evaluation Strategy.

Constraint C.7.1 (evaluation triggers). *Evaluation strategies must specify at least one evaluation trigger.*

```
context EvaluationStrategy inv evaluation_triggers :
onEvent−>size() + onValueChange−>size() +
          evaluationTime−>size() + onSituation−>size() > 0
```

Since we cannot use arithmetic operators, we split this constraint into a disjunction of four elementary constraints.

```
context EvaluationStrategy inv evaluation_triggers :
  Or( MultiplicityRestriction (EvaluationStrategy,onEvent,>,0),
      MultiplicityRestriction (EvaluationStrategy,onValueChange,>,0),
      MultiplicityRestriction (EvaluationStrategy,evaluationTime,>,0),
      MultiplicityRestriction (EvaluationStrategy,onSituation,>,0))
```

Constraint C.7.2 (cyclic evaluation). *Evaluation strategies must not, directly or indirectly, trigger their own evaluation.*

```
context EvaluationStrategy
def relatedSituations () =
  self.onSituation−>union(self.onSituation.relatedSituations())

inv cyclic_evaluation :
  self.relatedSituations()−>excludes(self.situationDefinition)
```

```
context EvaluationStrategy inv cyclic_evaluation :
  NoCyclicDependency(EvaluationStrategy, onSituation.evaluatedWhen)
```

Constraint C.7.3 (trigger eval). *The evaluation of a situation trigger can only be caused by incoming events, metric updates, or situation occurrences in a related monitoring context.*

```
context SituationDefinition inv trigger_eval :
 evaluatedWhen.onEvent.monitoringContextDefinition−>forAll(
 getRelatedContexts().situationDefinition −>includes( self ) ) and

 evaluatedWhen.onValueChange.monitoringContextDefinition−>forAll(
 getRelatedContexts().situationDefinition −>includes( self ) ) and

 evaluatedWhen.onSituation.monitoringContextDefinition−>forAll(
 getRelatedContexts().situationDefinition −>includes( self ) )
```

This constraint cannot be expressed using our constraint patterns because the structure of the *ForAll* pattern does not match the constraint.

Figure C.5 shows the RSA representation of the pattern instances. Out of 3 constraints in this section, 2 can be expressed using patterns.

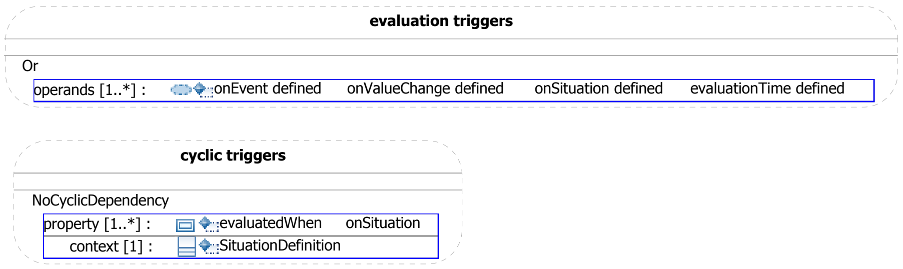


Figure C.5: Pattern instances for evaluation strategies.

C.8 Monitored Entities.

Constraint C.8.1 (slot values). *For each MonitoredEntity, a slotValue's targetSlotDefinitions must be owned by the monitoringContextDefinition of the MonitoredEntity.*

```
context MonitoredEntity inv slot_values:
  outputSlotValue.targetOutputSlotDefinition->forAll( s |
          monitoringContextDefinition. slotDefinition ->includes( s ) )
```

```
context MonitoredEntity inv slot_values:
  UniqueIdentifier(MonitoredEntity, outputSlotValue,targetOutputSlotDefinition)
```

Constraint C.8.2 (slot targets). *Different slotValues of MonitoredEntities must have different targetSlotDefinitions.*

```
context MonitoredEntity inv slot_targets :
self.outputSlotValue->forAll( v1,v2: OutputSlotValueSpecification |
         v1.targetOutputSlotDefinition = v2.targetOutputSlotDefinition implies v1 = v2 )
```

```
context MonitoredEntity inv slot_targets :
  UniqueIdentifier(MonitoredEntity,outputSlotValue.targetOutputSlotDefinition)
```

Constraint C.8.3 (slot coverage). *The slotValues must cover all KeyDefinitions of the monitoringContextDefinition of a MonitoredEntity.*

```
context MonitoredEntity inv slot_coverage:
self.monitoringContextDefinition.keyDefinition->forAll( k |
         outputSlotValue.targetOutputSlotDefinition->includes( k ) )
```

Since this constraint refers to the context element within the quantified statement, it cannot be expressed using our patterns; the *ForAll* pattern does not match this expression. Figure C.6 shows how the constraints from this subsection are represented in RSA. Out of three constraints in this section, two can be expressed using patterns.

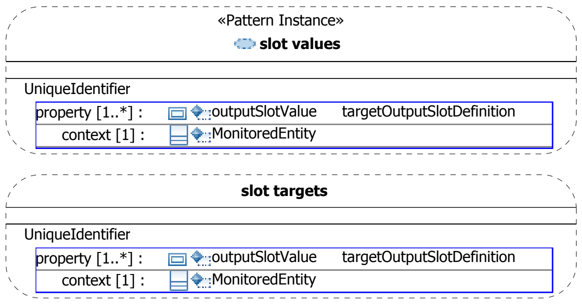


Figure C.6: Pattern instances for monitored entities.

C.9 Cyclic Dependencies.

Constraint C.9.1 (map cycles). *The directed graph defined by metrics and maps is acyclic.*

```
context MapDefinition
def allInputMaps(): self.input.mapDefinition->union(self.input.mapDefinition.allInputMaps())
inv map_cycles: self.allInputMaps()->excludes(self)
```

```
context MapDefinition inv map_cycles:
  NoCyclicDependency(MapDefinition, input.MapDefinition)
```

Constraint C.9.2 (context cycles 2). *The directed graph of monitoring context that are connected via context relations is acyclic.*

```
context MonitoringContextDefinition
def allChildContexts() = self.childContextRelationship.childContextDefinition->union(
        self.childContextRelationship.childContextDefinition.allChildContexts())
inv context_cycles: self.allChildContexts()->excludes(self)
```

```
context MonitoringContextDefinition inv context_cycles:
  NoCyclicDependency(MonitoringContextDefinition,
                     childContextRelationship.childContextDefinition)
```

Constraint C.9.3 (cyclic triggers). *The evaluation strategy of a situation may not depend on the situation itself.*

A violation of this constraint causes deadlocks.

```
context SituationDefinition
def relatedSituations() =
  self.evaluatedWhen.onSituation->union(self.evaluatedWhen.onSituation.relatedSituations())
inv cyclic_triggers :
  self.relatedSituations()->excludes(self)
```

```
context SituationDefinition inv cyclic_triggers :
  NoCyclicDependency(SituationDefinition, evaluatedWhen.onSituation)
```

Figure C.7 shows how the patterns from this subsection are represented in RSA. All three constraints in this section can be expressed using constraint patterns.

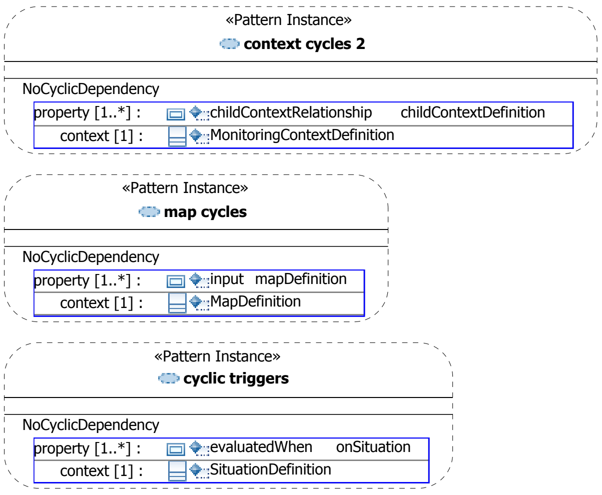


Figure C.7: Pattern instances for cyclic dependencies.

C.10 Unsupported Elements.

Constraint C.10.1 (no multi-valued metrics). *Multi-valued metrics are not supported.*

```
context MetricDefinition inv no_multi−valued_metrics:
  if  self.upperBound.oclIsTypeOf(Integer)
  then self.upperBound.value=1
  else true
  endif
```

```
context MetricDefinition inv no_multi−valued_metrics:
  IfThenElse(TypeRelation(MetricDefinition,upperBound,{Integer}),
             AttributeValueRestriction ( MetricDefinition ,upperBound,=,1),
            )
```

Constraint C.10.2 (no history). *The keepHistoryFor relation in metrics is not supported.*

```
context MetricDefinition inv no_history:
  self.keepHistoryFor−>size() = 0
```

```
context MetricDefinition inv no_history:
   MultiplicityRestriction  ( MetricDefinition ,keepHistoryFor,=,0)
```

Constraint C.10.3 (no external values). *External Value Specifications are not supported.*

```
context ExternalMetricDefinition inv no_external_values:
  ExternalMetricDefinition :: allInstances()−>isEmpty()
```

This constraint cannot be expressed using our constraint patterns.

Constraint C.10.4 (no data entries). *Data Entry Fields are not supported.*

```
context DataEntryFieldDefinition inv no_data_entries:
  DataEntryFieldDefinition :: allInstances()−>isEmpty()
```

This constraint cannot be expressed using our constraint patterns. Since this constraint shares a similar structure with Constraint C.10.3, we create a new constraint pattern in Section 8.2.4 to express these types of constraints.

Constraint C.10.5 (simple metrics). *There is no support for structured or multi-valued metrics.*

```
context MetricDefinition inv simple_metrics:
not (self.type.oclIsTypeOf(DataType) or
     if self.upperBound.oclIsTypeOf(Integer) then (self.upperBound.value>1) else false endif)
```

```
context MetricDefinition inv simple_metrics:
  Negation(Or(TypeRelation(MetricDefinition,type,{DataType}),
              IfThenElse(TypeRelation(MetricDefinition,upperBound,{Integer}),
                         AttributeValueRestriction ( MetricDefinition ,upperBound,=,1),
                         )
              )
           )
```

Constraint C.10.6 (no constraints). *The ownedConstraint association, inherited from NamedElement, should not be used in a SlotDefinition.*

```
context SlotDefinition inv no_constraints:
  self.oclAsType(NamedElement).ownedConstraint−>isEmpty()
```

```
context SlotDefinition inv no_constraints:
   MultiplicityRestriction ( SlotDefinition ,ownedConstraints,=,0)
```

Figure C.8 shows how the constraints from this subsection are represented in RSA. Out of six constraints in this section, four can be expressed using patterns. The figure also shows two more extracts from the monitor model's meta-model that we have left our earlier for conciseness reasons.

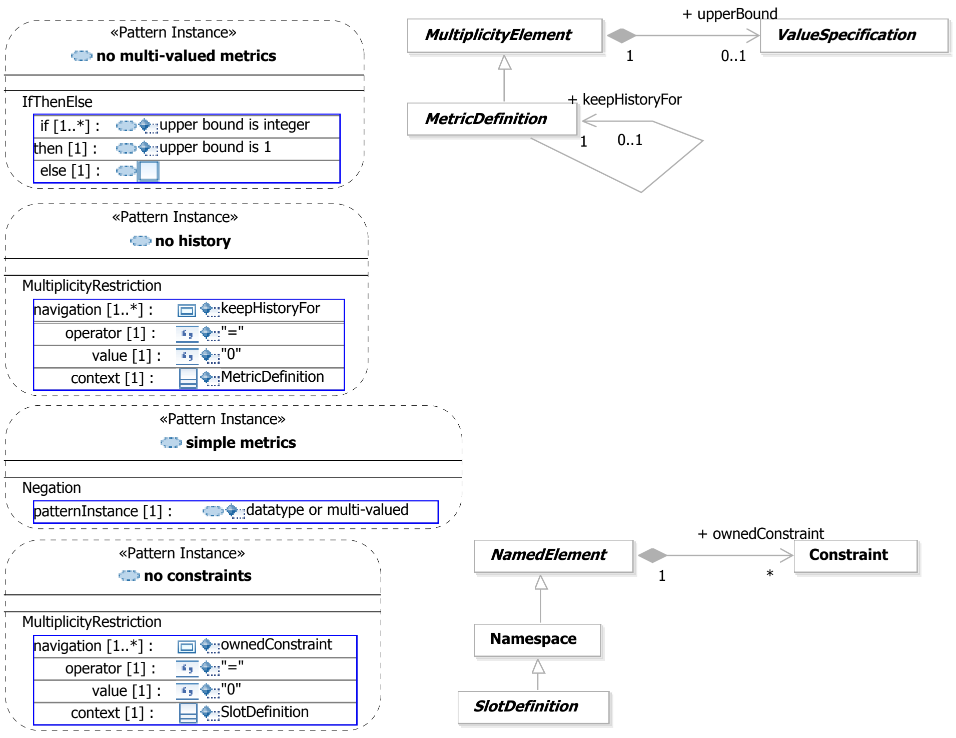


Figure C.8: Pattern instances against unsupported elements.

Appendix D

Acronyms

Acronyms

API	Application Programming Interface
CASE	Computer Aided Software Engineering
CPU	central processing unit
DSL	domain-specific language
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
FOL	first-order logic
GOF	Gang of Four
GUI	Graphical User Interface
HOL	higher-order logic
IT	information technology
JML	Java Modeling Language
KPI	key performance indicator
MDA	Model-Driven Architecture
MDD	Model-driven development
MDE	Model-Driven Engineering
MDT	Eclipse Model Development Tools
MOF	Meta Object Facility
MSFOL	many-sorted FOL
OCL	Object Constraint Language

RAS	Reusable Asset Specification
RSA	IBM Rational Software Architect
SAT	Boolean satisfiability problem
UML	Unified Modeling Language
VDM	Vienna Development Method
WBM	IBM WebSphere Business *Modeler*
MON	IBM WebSphere Business *Monitor*

List of Definitions

List of Figures

Curriculum Vitae

Personal Information

Michael Sebastian Wahler

Speerstrasse 47, 8038 Zurich, Switzerland

Date of Birth: March 10, 1978 (Munich, Germany)

Languages: German (native), English (fluent), French (basic)

Academic Record

March 2004 - February 2008

Swiss Federal Institute of Technology (ETH Zurich)
Doctoral studies under the supervision of Prof. David Basin

November 1998 - December 2003

Technical University of Munich, Germany
Diploma degree in computer science

Relevant Work Experience

March 2004 - February 2008

IBM Zurich Research Laboratory, Switzerland
Pre-doctoral researcher in the *Business Integration Technologies Group*.
Manager: Dr Jana Koehler

November 2000 - July 2003

Technical University of Munich, Germany
Student teaching assistant ("tutor")