

Scenario Testing using OWL

A dissertation submitted in fulfillment of the
requirements for the award of the degree

Master of Computer Science (Research)

from

UNIVERSITY OF KWAZULU-NATAL

by

Hendrina Francina Harmse

College of Agriculture, Engineering and Science

March 2015

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

Abstract

The main contribution of this dissertation is to provide an approach and related techniques and guidelines for an OWL 2 formalization of scenario testing, called formal scenario testing, that can be used to validate UML class diagrams. The main objective of formal scenario testing is to arrive at a complete and consistent conceptual schema (that can be expressed as a UML class diagram or directly in OWL 2), which is a cohesive and an accurate representation of the business domain. Hence, formal scenario testing is aimed at the validation rather than the verification of UML class diagrams. To this end, techniques are defined for formal scenario testing that can be used to validate the accuracy of UML class diagrams. Moreover, formal scenario testing can be used to validate the cohesiveness of UML class diagrams.

In service of this main contribution, the translation of UML class diagrams to DLs (and in particular $\mathcal{SROIQ}^{(D)}$) and OWL 2 are revisited. Firstly, a number of UML class diagram features have not been explicitly translated to $\mathcal{SROIQ}^{(D)}$ or OWL 2, for which the translation is made explicit. Secondly, some newer features of UML class diagrams have never before been translated to any DL. A notable example is ID constraints on classes. Lastly, some of the existing UML class diagram to DL/OWL 2 translations need to be refined for the purpose of formal scenario testing.

The formal scenario testing approach, and related techniques and guidelines, presented in this dissertation is of value to modellers since it enables them to validate UML class diagrams during the requirements engineering phase. Any endeavour that improves the detection and remedy of errors during the requirements engineering phase, helps to decrease the number of errors that are propagated to the later phases of the SDLC. Reducing the number of errors throughout the SDLC reduces the cost of the development of a software system.

Acknowledgments

First of all I will like to thank the Centre of Artificial Intelligence Research, University of KwaZulu Natal and CSIR Meraka Institute, South Africa for their financial assistance and infrastructure support. Without this support this research would not have been possible.

Secondly, I will like to thank my supervisors, Arina Britz, Auroa Gerber and Deshendran Moodley. Thank you for allowing me the freedom to explore the ideas I had, giving me guidance when needed and being excited about this research. Your optimism and thoughtful advice have made this onerous task an absolute pleasure.

Lastly, thank you to my partner for not only allowing me this time, but also for actively encouraging me to do my masters. Thank you for the many chats in which you enabled me to, at least partially, understand some of the challenges of the requirements engineering phase.

Contents

Abstract	iii
Acknowledgments	iv
1 Introduction	1
1.1 Requirements Engineering	1
1.1.1 Requirements Elicitation	2
1.1.2 Requirements Specification	2
1.1.3 Requirements Validation	3
1.2 Conceptual Modelling	3
1.2.1 Conceptual Modelling and Object-oriented Analysis	4
1.2.2 Theoretical Basis of Object-oriented Analysis	4
1.2.3 Object-oriented Analysis versus Object-oriented Design	5
1.2.4 Object-oriented Analysis versus Conceptual Data Modelling	5
1.3 Description Logics	6
1.3.1 Syntactic Building Blocks	6
1.3.2 Analogy between Description Logics and Object-orientation	6
1.3.3 Decidability and Complexity	7
1.3.4 Formalisms used in this Dissertation	7
1.4 The Case for the Formalization of Scenario Testing	8
1.4.1 DL Translations of UML Class Diagrams	8
1.4.2 Validating the Conceptual Schema	9
1.4.3 Formal Scenario Testing Intuitions	10
1.5 Aims and Objectives	10
1.6 Outline of Dissertation	11
2 UML Class Diagrams and Heuristics	13
2.1 UML Class Diagrams	13
2.1.1 Classes	13
2.1.2 Data Types	15

2.1.3	Binary Associations and Attributes Revisited	16
2.1.4	Generalization/Specialization of Classes	17
2.1.5	Association Specialization, Subsetting and Redefinition	18
2.1.6	Identity Constraints	20
2.1.7	Qualified Names	21
2.2	Modelling Heuristics	21
2.2.1	Separable Class Cohesion	22
2.2.2	Multifaceted Class Cohesion	22
2.2.3	Non-delegated Class Cohesion	25
2.2.4	Concealed Class Cohesion	27
2.2.5	Low Inheritance Cohesion of Attributes	27
2.2.6	Low Inheritance Cohesion of Operations	28
2.3	Summary	29
3	Description Logics and OWL 2	31
3.1	Description Logics	31
3.1.1	Description Logic Primer	31
3.1.2	Semantics	33
3.1.3	\mathcal{AL}	33
3.1.4	Extending DLs with Data Types	36
3.1.5	DL Nomenclature	38
3.1.6	Characteristics of DLs	38
3.2	DLs for the Translation of UML Class Diagrams	40
3.2.1	\mathcal{ALCQI}	40
3.2.2	$\mathcal{SROIQ}^{(D)}$	41
3.2.3	OWL 2	43
3.3	Reasoning Tasks	46
3.3.1	Deductive Reasoning Tasks	46
3.3.2	Other Reasoning Tasks	47
3.4	Summary	48
4	DL Translations of UML Class Diagrams	49
4.1	Classes	49
4.2	Attributes	50
4.2.1	Multiplicity	51
4.3	Operations	51
4.3.1	Operations with no Parameters	51
4.3.2	Operations with Parameters	52

4.4	Binary Associations	53
4.4.1	Multiplicity	53
4.5	Generalization/Specialization of Classes	54
4.6	Association Specialization	54
4.7	Data Types	55
4.8	Summary	55
5	OWL and DL Translations for Scenario Testing	60
5.1	UML Class Diagram Identity Constraints	61
5.1.1	Problematic Interpretation of Compound Keys	61
5.1.2	Identity Constraint Challenges and OWL 2 Easy Keys	62
5.1.3	The Effect of Easy Keys Compromises on Formal Scenario Testing	62
5.2	Tight Specification of Domain and Range Restrictions	63
5.2.1	Attributes	64
5.2.2	Binary Associations	65
5.2.3	Operations	65
5.3	Operations	66
5.3.1	Explicit Naming Convention	67
5.3.2	An Operation is Performed by the Class that Defines it	68
5.3.3	Operations with No Parameters	69
5.3.4	OWL 2 Translation of Operations	69
5.3.5	Unique Return Values	71
5.3.6	Operations with no Return Values	71
5.4	Translations for Modeller Productivity	73
5.4.1	Enumerations	73
5.4.2	Limiting Redundancy of Assertions for Binary Associations	74
5.5	Subsetting and Redefinition of Association Ends	75
5.6	On the Equivalence of Attributes and Binary Associations	75
5.7	A Note on Uniqueness of Names in UML Class Diagrams	76
5.7.1	Attributes and Associations	76
5.7.2	Operations and Parameters	77
5.7.3	Dealing with Anonymous Classes	78
5.8	Why Composition and Aggregation are excluded from Formal Scenario Testing	79
5.9	Contribution and Related Research	80
5.9.1	Contribution	80
5.9.2	Related Research	81
5.10	Summary	82

6	Formal Scenario Testing	92
6.1	Approach	92
6.1.1	Steps of the Formal Scenario Testing Approach	93
6.1.2	Key Characteristics of Formal Scenario Testing	93
6.2	Techniques	94
6.2.1	Consistent Scenario Tests	94
6.2.2	Inconsistent Scenario Tests	96
6.2.3	Classification Scenario Tests	97
6.2.4	Repairs for Formal Scenario Testing	102
6.3	Guidelines	103
6.3.1	Dealing with OWA	104
6.3.2	Dealing with UNA	106
6.3.3	Structuring Scenario Tests in Protégé	106
6.4	A Case Study	108
6.4.1	Business Domain	108
6.4.2	Deficiencies of a Naïve UML Class Diagram and a Solution	109
6.4.3	Validating the UML Class Diagram	111
6.4.4	Adoption and Preliminary Feedback	114
6.5	Contribution and Related Research	115
6.5.1	Contribution	115
6.5.2	Related Research	117
7	Applying Formal Scenario Testing	119
7.1	Separable Class Cohesion	120
7.1.1	Detection	120
7.1.2	Validation	123
7.2	Multifaceted Class Cohesion	125
7.2.1	Detection	125
7.2.2	Validation	126
7.3	Non-delegated Class Cohesion	127
7.3.1	Detection	127
7.3.2	Validation	129
7.4	Concealed Class Cohesion	131
7.4.1	Detection	131
7.4.2	Validation	138
7.5	Low Inheritance Cohesion of Attributes	141
7.5.1	Detection	141
7.5.2	Validation	142

7.6	Low Inheritance Cohesion of Operations	144
7.6.1	Detection	144
7.6.2	Validation	145
7.7	Contribution and Related Research	147
7.7.1	Contribution	147
7.7.2	Related Research	148
8	Conclusion	150
8.1	Contribution	150
8.1.1	The Research Gap Identified	150
8.1.2	How the Gap is addressed by this Research	150
8.1.3	The Value Proposition of this Research	151
8.1.4	Presentation and Publication in Support of this Dissertation	151
8.2	Future Research	152
8.3	Summary	152
	Appendices	153
A	RatesConfig Class Diagram Translated to OWL 2	154
B	Separable Class Cohesion Examples Translated to OWL 2	157
B.1	Translation of <code>Employee</code> Class with Separable Class Cohesion	157
B.2	Translation of Redesigned <code>Employee</code> Class	159
C	Multifaceted Class Cohesion Examples Translated to OWL 2	161
C.1	Translation of <code>ContactInformation</code> Class with Multifaceted Class Cohesion	161
C.2	Translation of Redesigned <code>ContactInformation</code> Class	162
D	Non-delegated Class Cohesion Examples Translated to OWL 2	163
D.1	Translation of <code>Employee</code> Class with Non-delegated Class Cohesion	163
D.2	Translation of Redesigned <code>Employee</code> Class	164
E	Concealed Class Cohesion Examples Translated to OWL 2	165
E.1	Translation of <code>LeaveRequest</code> and <code>PerformanceReview</code> Classes with Concealed Class Cohesion	165
E.2	Translation of Redesigned <code>LeaveRequest</code> and <code>PerformanceReview</code> Classes . .	167
F	Low Inheritance Cohesion Examples Translated to OWL 2	170
F.1	Translation of <code>Rectangle</code> and <code>Square</code> Classes with Low Inheritance Cohesion	170
F.2	Translation of Redesigned <code>Rectangle</code> and <code>Square</code> Classes	170
F.3	Translation of <code>Bird</code> Inheritance Hierarchy with Low Inheritance Cohesion . .	171

F.4 Translation of Redesigned Bird Inheritance Hierarchy	172
Bibliography	174

List of Figures

1.1	Relation of contributions of chapters 5–7.	11
2.1	Different graphical representations of classes.	14
2.2	An example of an enumeration data type.	15
2.3	Association A exists between class C and class T.	16
2.4	Different ways to depict an association with one association end.	16
2.5	Different ways to depict an association with two association ends.	16
2.6	Classes C1, C2, ..., Cn specializes class C.	18
2.7	Class diagrams illustrating association specialization, subsetting and redefinition.	19
2.8	Examples of identity constraints applied to classes.	20
2.9	Classes C1 and C2 both have an attribute called a.	21
2.10	Classes C1 and C2 both have an operation with name f.	21
2.11	The <code>Employee</code> class has separable class cohesion.	23
2.12	<code>pablo</code> and <code>projectX</code> are both instances of the <code>Employee</code> class.	23
2.13	Splitting the <code>Employee</code> class into <code>Employee</code> and <code>Project</code> classes.	23
2.14	Setting project related properties on the <code>Project</code> class.	23
2.15	The <code>ContactInformation</code> class has multifaceted class cohesion.	24
2.16	Instances of the <code>ContactInformation</code> class.	24
2.17	The redesign of <code>ContactInformation</code> class.	24
2.18	Instances of the redesigned <code>ContactInformation</code> class.	25
2.19	The <code>Employee</code> class has non-delegated class cohesion.	25
2.20	Instances of the <code>Employee</code> class.	26
2.21	Extracting the project concept into its own <code>Project</code> class.	26
2.22	Instances of the redesigned <code>Employee</code> and <code>Project</code> classes.	26
2.23	The classes <code>LeaveRequest</code> and <code>PerformanceReview</code> have concealed class cohesion.	27
2.24	Redesign of the <code>LeaveRequest</code> and <code>PerformanceReview</code> with <code>Period</code> class.	27
2.25	The <code>Square</code> and <code>Rectangle</code> classes have low inheritance cohesion.	28
2.26	A redesign of the <code>Square</code> and <code>Rectangle</code> classes.	28
2.27	The <code>Bird</code> class hierarchy have low inheritance cohesion.	29
2.28	A redesign of the <code>Bird</code> inheritance hierarchy.	29

4.1	Reification of a $(m+2)$ -ary relation.	52
6.1	Relation of contributions of chapters 5–7.	92
6.2	A UML class representing a robot.	95
6.3	An instance representing a red robot.	95
6.4	An instance of <code>Robot</code> that should be disallowed.	96
6.5	A redesign of the <code>Robot</code> class.	97
6.6	Explanations for a broken <code>Robot</code>	98
6.7	<code>Robot</code> and <code>TrafficLight</code> have the same attributes.	98
6.8	The classes <code>Robot</code> and <code>TrafficLight</code> are not equivalent.	100
6.9	An individual with properties <code>colour</code> and <code>lastMaintenanceDate</code>	101
6.10	The classes <code>ColourDomain</code> and <code>LastMaintenanceDateDomain</code> are equivalent.	101
6.11	An individual is inferred to be of type <code>Robot</code>	102
6.12	An individual is inferred to be of type <code>TrafficLight</code>	105
6.13	Explanation for the inconsistent scenario test in (6.8).	105
6.14	Explanation for a robot with no colour in (6.9).	106
6.15	The same TBox is reused for different ABoxes.	108
6.16	The initial UML class diagram for rate configuration.	109
6.17	An accurate representation of the business requirements.	110
6.18	Explanation for the disallowed scenario.	112
6.19	<code>interleadingHotelRateConfig</code> is of type <code>InterleadingHotelRateConfig</code>	113
6.20	<code>interleadingHotel</code> is of type <code>InterleadingHotelRateConfig</code>	114
7.1	Classification of the individual <code>pablo</code>	121
7.2	Classification of the individual <code>projectX</code>	121
7.3	Classification of the individual <code>sandy</code>	122
7.4	Classification of the individual <code>projectMassMarket</code>	123
7.5	The individual <code>projectX</code> is correctly classified.	124
7.6	The individual <code>projectMassMarket</code> is correctly classified.	124
7.7	Multifaceted class cohesion validated by inconsistency.	128
7.8	Non-delegated class cohesion validated by inconsistency.	130
7.9	Classification of the individual <code>individualUsingPeriodInfo</code>	133
7.10	Inferred equivalences.	134
7.11	Detecting the operations with the same signature.	135
7.12	Inferred class hierarchy for operation.	136
7.13	Classification of the individual <code>individualUsingPeriodInfo</code> for the redesign.	137
7.14	Inferred class hierarchy of the <code>FromDateDomain</code> class.	138
7.15	Validating the signature of an operation.	139
7.16	<code>individualUsingPeriodInfo</code> is classified as being of type <code>Period</code>	140

7.17	The individual <code>square</code> is classified as being of type <code>Rectangle</code>	142
7.18	Explanation of the inconsistency for <code>square</code>	142
7.19	Correct classification of the individual <code>quadrilateral</code>	143
7.20	Individual with <code>height</code> and <code>width</code> properties is classified as a <code>Rectangle</code> . . .	143
7.21	Classification based on the <code>fly()</code> operation yields the class <code>Bird</code>	144
7.22	The <code>Bird</code> class includes both the <code>Penguin</code> and <code>Eagle</code> classes.	145
7.23	Penguins not flying results in an inconsistency.	145
7.24	Only birds of type <code>FlyingBird</code> can fly.	146
7.25	Eagles are the only birds of type <code>FlyingBird</code>	146
7.26	All things of type <code>Bird</code> can walk.	147
7.27	Both eagles and penguins can walk.	147

Chapter 1

Introduction

In 1987 Frederick P. Brooks stated [22]:

“The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.”

The requirements specification Brooks refers to here, is created during the requirements engineering phase of the software development life-cycle (SDLC). As part of the requirements engineering phase, a conceptual schema can be created that represents the concepts, properties and relationships of the business domain. The work presented in this dissertation contributes to the objective, as stated by Brooks, by helping to achieve a complete and consistent conceptual schema. This is achieved by providing a description logic (DL) formalization of the industry standard approach for validating requirements called *scenario testing*. This DL formalization of scenario testing is called *formal scenario testing*.

The aim of this introductory chapter is to provide sufficient information, without delving too deeply into the details, to enable a succinct description of the contributions of this dissertation. This section begins by providing a wider perspective on the SDLC, and in particular the requirements engineering phase in Section 1.1. Key notions are clarified regarding conceptual modelling and DLs in Sections 1.2 (p. 3) and 1.3 (p. 6) respectively. In Section 1.4 (p. 8) an intuitive motivation is given for the formal scenario testing of conceptual schemas. Section 1.5 (p. 10) explicates the aims and objectives of this dissertation and Section 1.6 (p. 11) presents an outline of the chapters of this dissertation.

1.1 Requirements Engineering

The SDLC is a phased approach to delivering software with the objective of delivering specific artefacts at the end of each phase. At a high-level, irrespective of the exact SDLC model being applied (i.e. Waterfall, Spiral, etc), all the different SDLC models agree in terms of having a requirements engineering-, design-, implementation-, followed by a test phase [46]. In this dissertation the focus is on the requirements engineering phase. The requirements

engineering phase consists of the following processes: requirements elicitation, requirements specification and requirements validation [96]. Each of these processes is discussed in the subsequent subsections.

1.1.1 Requirements Elicitation

Before a new software system can be built, one has to have knowledge of the functions the software system needs to perform. Knowledge regarding the functions of the intended software system is gained through a requirements elicitation process [96]. Nuseibeh observes that the term “elicit” is preferred to “capture” to emphasize that information regarding the intended software system is not merely gathered, but rather, it is gained through a process of interpretation, analysis, modelling and validation [95]. Indeed, one of the challenges of requirements elicitation is that the stakeholders often find it difficult to articulate their requirements concisely [18, 95].

In order to facilitate the requirements elicitation process, stakeholders are encouraged to think through specific *use cases*. A use case is a set of actions performed by a software system to produce an observable result. Typically, a use case consists of a number of *scenarios* [18, 30, 46, 104]. According to Gomaa “[a] scenario is one specific path through a use case”. The main scenario of a use case describes the most common path through a use case and alternative scenarios describe the less-frequent paths through a use case [46].

Based on the information gained from use cases and their various scenarios, a requirements specification document is compiled. This is the topic of the next section.

1.1.2 Requirements Specification

The main purpose of the requirements specification document is to facilitate effective communication between the various stakeholders [95]. Stakeholders are people who have a vested interest in the system. Broadly speaking stakeholders can be categorized as people for whom the software system has financial impact (customers, business owners), as people who use the software system (users) and as people who are responsible for building and maintaining the software system (development team) [95, 96]. During the requirements specification process business owners and users are responsible for reaching agreement on the requirements of the system. The requirements specification document is the key deliverable of the requirements specification process and serves as the basis from which the development team will design, implement and test the system.

The requirements specification document defines both the functional and non-functional requirements of the software system. Functional requirements refer to functions that need to be performed by the software system while non-functional requirements refer to quality-of-service objectives (i.e. performance, response times, etc.) of the software system [46, 96]. This

dissertation is only concerned with the functional requirements of a software system.

The requirements specification document typically includes natural language descriptions (often in the form of use cases) and conceptual schemas of the functional needs of the system [18, 46, 96]. In this work the focus will be on the creation of a complete and consistent conceptual schema. Conceptual modelling and conceptual schemas are discussed in more detail in Section 1.2 (p. 3). For the moment it is important to note that a complete and consistent conceptual schema is indispensable in permitting economy of communication between stakeholders. A complete and consistent conceptual schema helps in building consensus and resolving conflicts between business owners and users while giving concise guidance to the development team as to what functions the software system has to fulfill [48, 93, 95, 115]. It is therefore of the utmost importance to ensure the correctness of the conceptual schema, which provides the motivation for a requirements validation process.

1.1.3 Requirements Validation

Addressing software system defects earlier in the SDLC is more cost effective than addressing them later [34]. Resolving defects when a software system is in production is estimated to be 100 times as expensive when compared to resolving them during the requirements engineering- and design phases of the SDLC [16]. Additionally, poor requirements are often mentioned as the main reason for failure [101] and often the most expensive software failures have their roots in the business requirements [75].

It is due to these reasons that the requirements validation process forms an essential part of the requirements engineering phase. One manner which is often used to validate requirements is *scenario testing*. Formally, a scenario test is a test based on a scenario [18, 30, 46, 67, 95]. In this dissertation a formal DL formalization of scenario tests will be used to verify that the conceptual schema is consistent and validate that it corresponds with the requirements as specified by the stakeholders.

1.2 Conceptual Modelling

In this section a brief introduction to conceptual modelling and the core definitions, that are relevant to making the precise contributions of the current dissertation explicit, are provided. For conceptual modelling the focus will be on object-oriented analysis for which the terminology is clarified in Section 1.2.1 (p. 4). In Section 1.2.2 (p. 4) the theoretical basis that underpins object-oriented analysis is explained. Sections 1.2.3 (p. 5) and 1.2.4 (p. 5) are aimed at discerning aspects of object-oriented conceptual modelling that are often cause for confusion.

1.2.1 Conceptual Modelling and Object-oriented Analysis

The precise meaning of “conceptual modelling” is ambiguous [33, 72]. This is in part due to the inconsistent use of the term “conceptual model” [40, 42, 72, 96] and variances in terminology related to conceptual modelling (see for instance [48, 96]). In an attempt to limit ambiguity, this dissertation will use the conceptual modelling terminology as is defined by Mylopoulos [93] and Olive [96]. *Conceptual modelling* is the act of documenting a domain to facilitate understanding and communication between stakeholders [48, 93, 96, 115]. The artefact resulting from the conceptual modelling activity is called the *conceptual schema* [93, 96]. A conceptual schema is an abstraction of the conceptually relevant aspects of a domain from a particular point of view [18, 43, 82, 96]. Implementation specific details are excluded from a conceptual schema [96, 112]. A particular domain may be described by a number of different conceptual schemas [96].

A conceptual schema is constructed using a *conceptual modelling language* [96, 115]. Each conceptual modelling language has a particular syntax and semantics [115]. The syntax and semantics of a conceptual modelling language depends on a commitment to view a domain in a particular way. This commitment to view a domain in a particular way is called a *conceptual model* [96].

As an example, a domain can be described in terms of concepts, properties and relationships. This represents one conceptual model of a domain. Another conceptual model is to describe a domain as consisting of facts that can be either true or false. The same conceptual model can be used to describe different domains and the same domain can be described by different conceptual models [96].

The field of software engineering makes the fundamental assumption that a domain consists of a number of objects, which have properties and relationships between them, which are classified into concepts [96]. That is, the assumption is that the conceptual model consists of concepts, properties and relationships. This kind of conceptual modelling is often referred to as object-oriented analysis [18, 42, 43, 82, 104]. The conceptual modelling language most commonly adopted for object-oriented analysis is UML and in specific UML class diagrams [13, 112].

1.2.2 Theoretical Basis of Object-oriented Analysis

Classification is the core activity of object-oriented analysis [18, 96, 117]. Classification is the means via which people order knowledge according to the similarities they recognize between different objects they observe in the world. The specific classification approach that is applied when doing object-oriented analysis is called classical categorization [18]. Classification, in specific classical categorization, does not pertain to object-orientation alone, but rather, it reflects how people think in general about the world [80]. Olive explains the need and use of

classification as follows [96]:

“Classification provides cognitive economy because it allows us to structure knowledge about objects into two levels: concept and instance. At the concept level, we find the properties (both defining and nondefining) common to all instances of the concept. At the instance level, we find only the concept of which the object is an instance, and the particular properties of that instance. In the absence of classification, we would have to associate every instance with all of its properties. Classification reduces the amount of information we have to remember, communicate, and process; the extent to which it is reduced depends on the number of properties of the concept.”

It is precisely this cognitive economy, provided by a complete and consistent object-oriented conceptual schema, that is the essence of enabling efficient communication between stakeholders and a key objective of this dissertation.

1.2.3 Object-oriented Analysis versus Object-oriented Design

UML is a general-purpose modelling language designed for use in the analysis, design and implementation of object-oriented software systems [18, 43, 64, 82]. Object-oriented analysis is distinguished from object-oriented design in that the focus of object-oriented analysis is to elicit and describe the classes and objects that form part of the problem domain, while the focus of object-oriented design is on designing the software solution consisting of objects and related collaborations that would fulfill the requirements [18, 42, 43, 52, 82, 89, 104, 109]. Thus, object-oriented analysis is strongly related to conceptual modelling while object-oriented design is strongly related to the implementation details of realizing the conceptual schema in code.

1.2.4 Object-oriented Analysis versus Conceptual Data Modelling

Traditional software engineering clearly separated data from the operations that operate on the data [109, 117]. Conceptual data modelling aims to represent an abstraction of data [29, 113]. This is in contrast with object-oriented analysis where objects consists of the data and the operations that operate on the data [109]. Therefore, in this dissertation the use of conceptual modelling will be favoured to conceptual data modelling with the implicit understanding that conceptual modelling includes the operations that operate on the objects.

1.3 Description Logics

Description logics (DLs) are syntactic variants of first-order logic that are specifically designed for the conceptual representation of an application domain in terms of concepts and relationships between concepts [13, 24]. In Section 1.3.1 (p. 6) the building blocks of DLs are defined. In Section 1.3.2 (p. 6) the relation between object-orientation and DLs are made explicit. Important considerations in DL research are decidability and complexity, which are discussed in Section 1.3.3 (p. 7). In Section 1.3.4 (p. 7) the motivation is given for the DLs that will be used in this dissertation.

1.3.1 Syntactic Building Blocks

Expressions in DLs are constructed from *atomic concepts* (unary predicates), *atomic roles* (binary predicates) and *individuals* (constants). Complex expressions can be built inductively from these atomic elements using *concept constructors*. Formally a concept represents a set of individuals and a role a binary relation between individuals [8, 94].

Formally every DL ontology consists of a set of axioms that are based on finite sets of concepts, roles and individuals [78]. Axioms in a DL ontology are divided into the TBox, the RBox and the ABox. A TBox is used to define concepts and relationships between concepts (that is the terminology or taxonomy) and an ABox is used to assert knowledge regarding the domain of interest (i.e. that an individual is a member of a concept). Depending on the expressivity of the DL used, an ontology may include an RBox. An RBox is used to define relations between roles as well as properties of roles [8, 78, 94, 103].

1.3.2 Analogy between Description Logics and Object-orientation

The analogy between DLs and object-orientation can be observed when it is considered that the basic task in constructing a terminology is classification [8, 94]. Explicit subsumption relationships between concepts can be defined in the TBox. In object-orientation this can be achieved by definition of an inheritance hierarchy between classes. Classification is further solidified as the basis of DLs in that the core reasoning capabilities they provide are subsumption and instance checking. Subsumption computes a subsumption hierarchy, which essentially categorizes concepts into superconcept/subconcept relationships. Instance checking verifies whether a given individual is an instance of a specific concept [94].

In object-orientation the domain of interest is described in terms of classes that have properties, which are defined via attributes and/or associations. Objects that are classified by a class are called instances of the class [18, 64]. The analogy with DLs is that classes, attributes/associations and instances (or sometimes called objects) correspond respectively with concepts, roles and individuals in DLs [13, 24].

1.3.3 Decidability and Complexity

A feature of DLs is that they have decidable reasoning algorithms for standard reasoning tasks (i.e subsumption and instance checking) [6, 103]. Indeed, a fundamental goal of DL research is to preserve decidability to the point that decidability is considered to be a precondition for claiming that a formalism is a DL [103]. Standard DL reasoning algorithms are sound and complete and, even though the worst-case computational complexity of these algorithms are ExpTime and worse, in practical applications they are well-behaved [6].

The expressivity of a DL is determined by the number of different concept constructors it permits with a higher number of concept constructors in general correlating to higher expressivity. An important trade-off in DL design is to strike a balance between expressivity and computational complexity, since the more expressive a DL is, the higher is its computational complexity. Early DL research has focused on the search for DLs with tractable reasoning for core reasoning tasks, that is reasoning that can be completed in polynomial time. As mentioned before, in practice it turns out that even DLs with computational complexity of ExpTime and worse are well-behaved [6].

Tractability is of special importance when reasoning across large ontologies are required while for small ontologies it is less of a concern [28]. Formal scenario testing, as presented in this dissertation, will only deal with relatively small ontologies. As such preference can be given to DLs with high expressivity at the cost of high computational complexity.

1.3.4 Formalisms used in this Dissertation

In this dissertation the DLs *ALCQI* and *SROIQ^(D)* will be considered. *ALCQI* is one of the DLs used by Berardi, et. al. for the translation of UML class diagrams [13]. The contributions of this dissertation will mainly be focused on *SROIQ^(D)* and OWL 2. There are a number of reasons for this decision:

1. The formal semantics of OWL 2 is based on *SROIQ^(D)* [47]. OWL 2 is a World Wide Web Consortium (W3C) endorsed specification for the construction of ontologies for the semantic web [92].
2. As part of formal scenario testing, there is a requirement to be able to represent key constraints formally. OWL 2 semantics extends *SROIQ^(D)* with a relaxed form of DL-safe rules, which permits modelling of key constraints [97].
3. The ontologies that will be considered in formal scenario testing are relatively small. Since tractability is not a major concern for small ontologies [28], the DL *SROIQ^(D)* can be used, which has a complexity of N2ExpTime-complete. [47, 68].
4. The standardization of OWL 2 has resulted in the development of ontology editors and reasoners in support of OWL 2. The main open source ontology editors are Protégé

and Swoop [47]. A commercially available tool in this regard is TopBraid Composer (Meastro Edition) from TopQuadrant [2]. Hermit and Pellet are reasoners that support reasoning in $\mathcal{SROIQ}^{(D)}$ as well as SWRL [35]. SWRL serves as the basis of DL-safe rules [76], which is a requirement for reasoning on key constraints in OWL 2 [97].

1.4 The Case for the Formalization of Scenario Testing

Due to the potential for substantial cost savings (as explained in Section 1.1.3 on p. 3), a considerable amount of research has been done to mathematically formalize and verify conceptual schemas, and in specific, UML class diagrams. This research primarily focuses on DL formalizations of UML class diagrams. With regards to existing DL translations of UML class diagrams, Section 1.4.1 (p. 8) lists the kind of errors that can be detected, while Section 1.4.2 (p. 9) lists the kind of errors that cannot be detected. In Section 1.4.3 (p. 10) the key difference between formal scenario testing and existing approaches is explained.

1.4.1 DL Translations of UML Class Diagrams

Cali, et al. [24] and Berardi, et al. [13] laid the foundation for describing UML class diagrams in DLs. Cali, et al. described UML class diagrams in the DL \mathcal{DLR}_{ifd} [24]. Berardi, et al. extended this work by describing UML class diagrams in the DLs \mathcal{DLR}_{ifd} and \mathcal{ALCQI} [13]. Recent research described UML class diagrams in OWL 2 [118]. Based on these DL translations a number of reasoning tasks are possible on UML class diagrams [13]. The reasoning tasks that are most pertinent to the current dissertation are briefly explained next.

Consistency of the complete UML class diagram can be proved. This implies that at least one class in the diagram can be instantiated without violating any of the constraints defined in the diagram. Proving that a complete class diagram is consistent proves that it does not contain any contradictions.

Consistency of a single class in the diagram can be proved. This means that the class can be instantiated. A class that can never be instantiated indicates a design mistake within the diagram.

Classes in the UML class diagram can be classified to show inferred inheritance relationships. This information can be used to make inferred inheritance relationships that are not explicitly stated in the diagram explicit. It can also be used to detect low inheritance cohesion: that is, classes that are in the same inferred inheritance hierarchy, which are not suppose to share attributes and/or associations. Low inheritance cohesion indicates that the UML class diagram contains one or more modelling errors.

Redundant classes can be detected by proving that different classes in the diagram are indeed equivalent. Equivalent classes indicate classes that can be merged into a single class.

Formal reasoning on a UML class diagram can be used to detect the implicit consequences of the constraints enforced by the classes, associations and inheritance hierarchies contained in the diagram. A core objective of a conceptual schema is to facilitate communication between stakeholders. Any implicit consequences that are not explicitly stated in the diagram detracts from this objective.

1.4.2 Validating the Conceptual Schema

Existing research regarding the formalization of UML class diagrams are concerned with *verification* whereas formal scenario testing is concerned with *validation*. With regards to software requirements, Barry Boehm defined verification and validation informally as [17]:

Verification: “Am I building the product right?”

Validation: “Am I building the right product?”

DL verification of UML class diagrams can prove that a class diagram is free of logical inconsistencies. However, even when a class diagram is consistent, it may still not agree with the business requirements. When the conceptual schema is the right conceptual schema, it must give a *cohesive* and an *accurate* representation of the business requirements. The term “cohesive” here is used to capture two related notions. Firstly, when used in the context of a single class, it is meant that the class represents a single semantically meaningful concept. In object-oriented parlance, this is referred to as *model cohesion*, which is the most stringent type of *class cohesion* [21]. Secondly, when used in the context of a class hierarchy, it means that from a conceptual modelling perspective, it forms a generalization hierarchy. This is known as *inheritance cohesion*, which means that classes in a single class hierarchy have shared meaning. Inheritance cohesion is highest when the meaning of each class in the hierarchy is more specialized than that of its parent, for classes that have parents. When unrelated classes are found in the same class hierarchy, the class hierarchy displays low inheritance cohesion [38]. When a class diagram is “accurate”, it is not possible to instantiate the class diagram, or a single class in the diagram, such that it serves as a counter-example of the requirements of the business.

An important difference between verification and validation, from a conceptual modelling perspective, is that verification can be mathematically proven while validation cannot be mathematically proven. A conceptual schema can be verified by proving that it is free of logical inconsistencies. However, a conceptual schema cannot be validated mathematically. Rather, at most, techniques and heuristics can be provided, which can be used as evidence that the conceptual schema is potentially the right conceptual schema. In this dissertation, formal scenario testing will be the approach that underpins techniques and heuristics with regards to the validation process. This is discussed in the next section.

1.4.3 Formal Scenario Testing Intuitions

In formal scenario testing UML class diagrams are validated by giving a domain expert the opportunity to define scenarios that are either allowed or disallowed for the given business domain. The domain expert can define, in object-oriented parlance, a combination of instances, or in DL parlance, a set of individuals, that represents a single scenario. An allowed scenario indicates a real-world situation that is a requirement of the business domain. A disallowed scenario is a situation that should never be possible in the given business context. With the formal scenario testing approach presented here, scenarios can be validated by showing that the DL translation of a scenario is consistent for an allowed scenario and inconsistent for a disallowed scenario. Furthermore, classification is employed extensively to detect potential class cohesion and inheritance cohesion violations. This is made possible by the domain expert providing exemplars of the business domain.

Thus, in addition to considering UML class diagrams, formal scenario testing specifically includes instances. These instances can be represented in an object diagram. The UML specification is not overly prescriptive and allows objects to be present in UML class diagrams and class definitions to be present in UML object diagrams [64]. In this dissertation the assumption is made that UML class diagrams only contain classes while UML object diagrams only contain objects. With this assumption in place, it is noted that, loosely speaking (due to the UML specification not being restrictive), a TBox¹ corresponds with a UML class diagram and an ABox corresponds with a UML object diagram. Prior research on reasoning on UML class diagrams using DLs has focused on reasoning on UML class diagrams (respectively TBoxes) alone, whilst the formal scenario testing approach includes reasoning on UML object diagrams (respectively ABoxes) [24, 13, 118].

1.5 Aims and Objectives

The main contribution of this dissertation is to provide an approach and related techniques and guidelines for an OWL 2 formalization of scenario testing, called formal scenario testing, that can be used to validate UML class diagrams. The main objective of formal scenario testing is to arrive at a complete and consistent conceptual schema (that can be expressed as a UML class diagram or directly in OWL 2), which is a cohesive and an accurate representation of the business domain. Hence, formal scenario testing is aimed at the validation rather than the verification of UML class diagrams. To this end, techniques are defined for formal scenario testing that can be used to validate the accuracy of UML class diagrams. Moreover, formal scenario testing can be used to validate the cohesiveness of UML class diagrams.

In service of this main contribution, the translation of UML class diagrams to DLs (and

¹An RBox may be included if the expressivity of the underlying DL permits RBoxes.

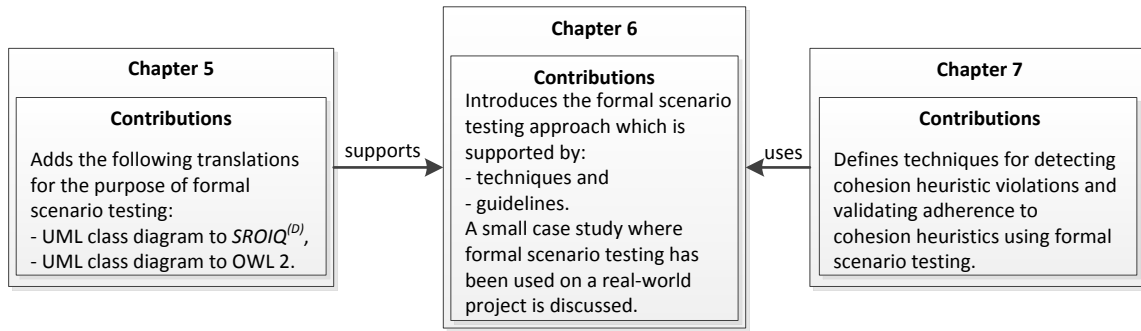


Figure 1.1: Relation of contributions of chapters 5–7.

in particular $SROIQ^{(D)}$) and OWL 2 are revisited. Firstly, a number of UML class diagram features have not been explicitly translated to $SROIQ^{(D)}$ or OWL 2, for which the translation is made explicit. Secondly, some newer features of UML class diagrams have never before been translated to any DL. A notable example is ID constraints on classes. Lastly, some of the existing UML class diagram to DL/OWL 2 translations need to be refined for the purpose of formal scenario testing.

The formal scenario testing approach, and related techniques and guidelines, presented in this dissertation is of value to modellers since it enables them to validate UML class diagrams during the requirements engineering phase. Any endeavour that improves the detection and remedy of errors during the requirements engineering phase, helps to decrease the number of errors that are propagated to the later phases of the SDLC. Reducing the number of errors throughout the SDLC reduces the cost of the development of a software system.

1.6 Outline of Dissertation

Chapters 2–4 represent the literature study in support of the contributions of this dissertation. Chapters 5–7 represent the contributions and Chapter 8 the conclusion of the current dissertation. How the contributions of chapters 5–7 relate to each other are depicted in Figure 1.1. A brief description of what is covered in each chapter is given below.

Chapter 2 : Key features of UML class diagrams are introduced. Related object-oriented design heuristics are discussed, which are helpful in evaluating the cohesiveness of classes and class hierarchies in a UML class diagram.

Chapter 3 : An overview of DLs is given, which is followed by the syntax and semantics of the DLs $ALCQI$ and $SROIQ^{(D)}$. The syntax and semantics of OWL 2 are discussed based on the semantics of $SROIQ^{(D)}$. Reasoning tasks that are of importance to the current dissertation are also discussed.

Chapter 4 : Existing translations of UML class diagrams to *ALCQI* and OWL 2 are discussed.

Chapter 5 : Not all features of UML class diagrams have been formalized in DLs or OWL 2 as yet. Here UML class diagram features that are pertinent to formal scenario testing are translated to *SROIQ^(D)* and OWL 2.

Chapter 6 : The formal scenario testing approach, the main contribution of this dissertation, is introduced. Techniques and guidelines used in formal scenario testing are defined. Preliminary feedback is given on a small case study where formal scenario testing has been used on a real-world project.

Chapter 7 : In this chapter techniques for detecting and validating object-oriented cohesion heuristics are defined using formal scenario testing.

Chapter 8 : A synopsis is given of the contributions of this dissertation and the value of these contributions is made explicit. A view is given on possible future directions of research; with regards to the topics that were investigated in this study.

Chapter 2

UML Class Diagrams and Heuristics

This chapter provides background on the aspects of UML class diagrams that are used throughout this dissertation (Section 2.1). Knowledge of the syntax and semantics of UML class diagrams is imperative for creating conceptual schemas expressed as UML class diagrams. However, this knowledge is not sufficient to enable a modeller to create complete and consistent conceptual schemas. This fact is attested by the vast literature dedicated to various heuristics for guiding the modelling of classes (see for instance [18, 46, 88, 89, 81, 116]). Formal scenario testing can be an aid in applying some of these heuristics. Therefore modelling heuristics are the topic of Section 2.2 (p. 21).

2.1 UML Class Diagrams

This section discusses key features of UML class diagrams that are used in formal scenario testing. In general UML is a large and complex conceptual modelling language [64, 104]. As a means to cope with the complexity of UML, Booch, et. al. [18] and Rumbaugh, et. al. [104] recommend modellers give preference to the essential elements of notation and only use advanced notation when their use is imperative to conveying meaning. This is the guideline that is followed in this dissertation.

2.1.1 Classes

A class denotes a set of instances which share the same attributes and operations [64]. From a classical categorization theory perspective, instances that have the attributes and are able to perform the operations defined by a class, are considered to be instances of the class [18].

Graphically a class is represented as a rectangle that consists of one to three compartments (see Figure 2.1). The first compartment denotes the name of the class and is compulsory. The second compartment lists the attributes and the third compartment the operations of the class. The second and third compartments can be suppressed as is shown in Figure 2.1(a). This is a convenient form of representation when there is a mere need to refer to a class without giving consideration to its attributes and operations. Figure 2.1(b) illustrates a class with

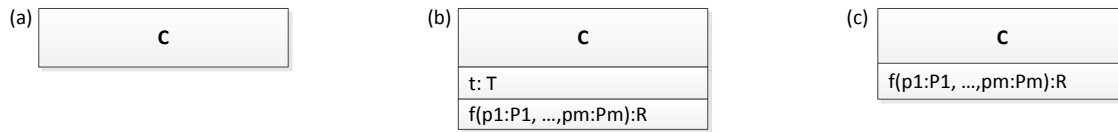


Figure 2.1: Different graphical representations of classes.

both attribute and operation compartments populated and Figure 2.1(c) shows a class with only the operation compartment populated [64].

Attributes

Attributes are used in UML class diagrams to define properties that are characteristic of a particular class [18, 64, 104, 117]. For example it is natural to accept that “having feathers” is a characteristic property of birds but not of dogs, while “having fur” is a characteristic of dogs but not of birds.

Figure 2.1(b) defines the attribute **t** of type **T** as belonging to class **C**. **T** can either denote a class or a data type. Data types are discussed in Section 2.1.2 (p. 15).

Operations

Throughout this dissertation UML class diagrams are considered from a conceptual modelling perspective, meaning that operations are therefore considered from a conceptual modelling perspective. Hence, the preference for the term “operation” rather than the term “method”. The distinction is that operations refer to the conceptual modelling of behaviour whereas methods refer to the implementation details of behaviour [104]. In UML class diagrams operations are used to define behaviours that are considered to be characteristic of a given class [18, 64, 104, 117]. As an example, the ability to fly is a behaviour that is associated with birds, but it is not a behaviour that is associated with for instance dogs. Using operations, it is possible to, for example, write a game in which instances of the **Bird** class can fly from point A to point B (on a screen). This behaviour should be distinguished from the attribute of “can fly”, which merely states that instances of the **Bird** class can fly, without being able to move said instance from point A to point B¹.

In Figures 2.1 (b) and (c) the operation **f(p1:P1, . . . , pm:Pm):R** is defined. This states

¹ From a DL perspective it may, for the simple case of the operation **fly()** that takes no parameters and returns nothing, seem as if there is no difference between the attribute “can fly” and the operation “fly”. Indeed, based on the translation of attributes in Section 5.2.1 (p. 64) and the translation of operations with no parameters and no return value in Section 5.3.6 (p. 71), the underlying DL representations of these are quite similar. However, from the perspectives of object-orientation and UML class diagrams, the attribute “can fly” is quite different from the operation “fly”. Whereas the value of the “can fly” attribute can be used to decide various rules, it will not by itself be able to cause instances of a **Bird** class to move from point A to point B (as in the game example). In order for instances of the **Bird** class to be able to fly from point A to point B, the **Bird** class has to be equipped with a “fly” operation.

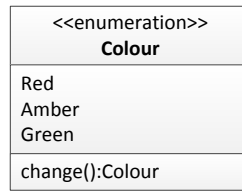


Figure 2.2: An example of an enumeration data type.

that the operation with name f takes parameters p_1, \dots, p_m respectively of type P_1, \dots, P_m returning a value of type R . An operation f that does not take any parameters or return any value is defined simply as $f()$.

2.1.2 Data Types

Data types differ from classes in that instances of data types are completely defined by their values. In particular, if two instances of a data type have the same value, they do in actual fact represent the same instance. Data types are similar to classes in that they can also have attributes and operations. However, the attributes and operations of data types do not define characteristic features of the data type. That is, instances that do have the attributes and operations defined by a data type do not imply that the instances are members of the data type. Rather, the value of the data type determines membership. Attributes are used to model the internal structure of a data type. UML distinguishes two types of data types namely primitive types and enumerations [64].

Primitive types are predefined types that contain no internal structure. The primitive types defined in UML are Boolean, Integer, UnlimitedNatural, Real and String [63, 64]. Boolean represents the logical values (*true*, *false*). Integer, UnlimitedNatural and Real respectively represent the mathematical concepts of integer, natural and real numbers. String represents sequences of characters [63].

The values that represent the values of an enumeration are explicitly defined in the conceptual schema. An example is the colours of a robot (traffic light), which can only consist of the values “red”, “amber” and “green”. This is illustrated in Figure 2.2. Also, as shown in Figure 2.2, data types can have operations. For instance the operation `change():Colour` can be used to model the behaviour of the colour of a robot changing. Enumerations can also have attributes, though this have not been indicated in the example.

In this dissertation only predefined primitive data types and enumerations are used. User-defined data types are not used due to scope constraints.

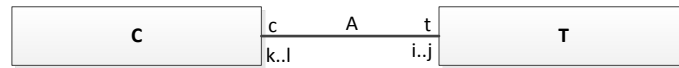


Figure 2.3: Association A exists between class C and class T.

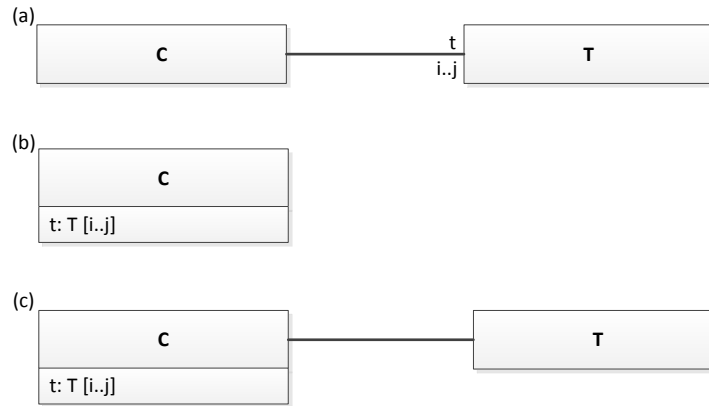


Figure 2.4: Different ways to depict an association with one association end.

2.1.3 Binary Associations and Attributes Revisited

Binary associations are used to define relations between two classes and/or datatypes. Associations between the instances of the classes are called links [64]. Figure 2.3 defines the association named A between the class C and the class T. A is the name of the association and c and t represent the names of the association ends. Usually either the name of the association or the names of the association ends is supplied [64].

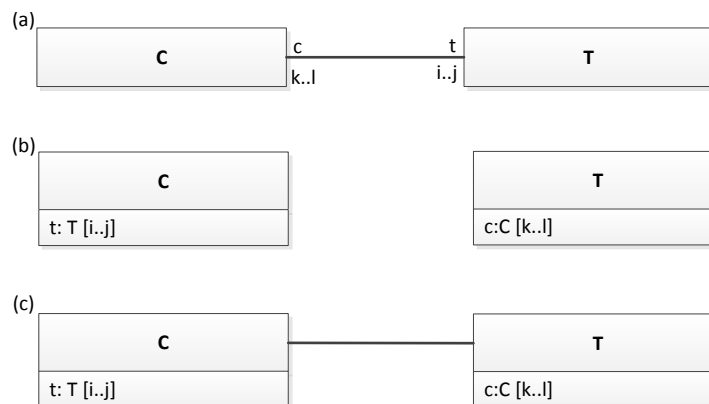


Figure 2.5: Different ways to depict an association with two association ends.

Equivalence of Attributes and Binary Associations Ends

Associations and attributes are closely related. The association end t of Figure 2.4(a) is equivalent to an attribute t of type T belonging to class C as is shown in Figure 2.4(b)². It is also possible to use the attribute and association notation together as in Figure 2.4(c) [64]. In this dissertation the graphical notation of Figure 2.4(c) is adopted.

For completeness sake Figure 2.5 shows that an association between classes C and T with two association ends c and t is equivalent to an attribute $t:T$ belonging to class C and an attribute $c:C$ belonging to class T .

Multiplicities

A multiplicity on an association end denotes the number of instances that can be associated with an instance of the opposite class. The multiplicity is expressed as a range $i..j$ where i is the lower bound and j the upper bound. Thus, Figure 2.4(a) on p. 16 states that i to j instances of T can be associated with an instance of class C . Similarly, a multiplicity on an attribute indicates the number of instances the attribute can consist of. Thus, for Figure 2.4(b) on p. 16 the attribute t can consist of i to j instances of type T [64].

When only one value is supplied for a multiplicity (rather than a range) it is assumed to be the upper bound. When no multiplicity is given the multiplicity $1..1$ or 1 is assumed. When the upper bound is unlimited it is denoted as $*$, however when it is used on its own (without a lower bound), its assumed meaning is $0..*$ [64].

2.1.4 Generalization/Specialization of Classes

Generalization defines a taxonomic relation between a more general parent class and a more specialized child class. Or stated differently: a child class specializes a more general parent class. The meaning of inheritance is such that every instance of a child class is per definition an instance of the parent class, but not every instance of the parent class is necessarily an instance of a child class [64].

Figure 2.6 indicates that the classes C_1, C_2, \dots, C_n specializes class C . An inheritance hierarchy can be annotated with a `{covering, disjoint}` annotation. Permitted values for `covering` are `complete` and `incomplete` where `complete` indicates that every instance of the parent class is also an instance of at least one child class and `incomplete` indicates that every instance of the parent class is not necessarily an instance of a child class. That is, there are instances of the parent class that are not specialized by a child class. `disjoint` indicates whether an instance can belong to more than one child class. Permitted values

²This is a slight simplification of associations since the ends of an association can be owned by the partaking classes or the association. An association end and an attribute are only equivalent for an end that belongs to the class [64]. In this dissertation the assumption is that association ends belong to the class.

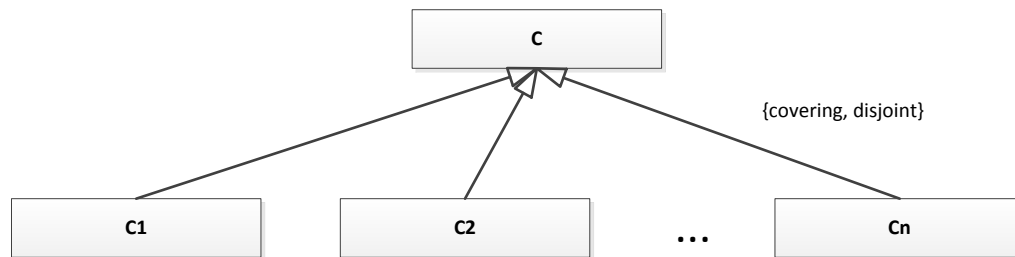


Figure 2.6: Classes C1, C2, ..., Cn specializes class C.

for `disjoint` are `overlapping` and `disjoint`. `overlapping` indicates that an instance can belong to more than one child class while `disjoint` states that an instance can at most belong to one child class. When an inheritance hierarchy is not explicitly annotated with a `{covering, disjoint}` annotation, the default annotation `{incomplete, disjoint}` is assumed to apply [64].

2.1.5 Association Specialization, Subsetting and Redefinition

With regards to associations and attributes the UML specification defines three closely related notions namely association specialization, subsetting and redefinition. Association specialization is applied at the level of the complete association while subsetting and redefinition are applied at the level of association ends and attributes [64].

The meaning of association specialization is similar to that of classes. That is, association A2 is a specialization of association A1 in Figure 2.7(a). Therefore every link between instances of classes C3 and C4 is necessarily a link between instances of classes C1 and C2. However, every link between instances of classes C1 and C2 is not necessarily a link between instances of classes C3 and C4.

In Figure 2.7(b) it is stated that `c4 {subsets c2}`. This means that the collection of instances represented by the association end `c4` is a subset of the collection of instances represented by the association end `c2`. The same holds for association ends `c3` and `c1`. Subsetting is distinguished from association specialization in that subsetting considers set membership only whereas association specialization specializes the characteristics that determine link membership.

Redefinition is used to change the definition of a feature. As an example association end `c4` redefines `c2` in Figure 2.7(c). Redefinition is distinguished from association specialization in that redefinition is defined for an association end rather than for the complete association.

The intuition that these notions are rather closely related is confirmed by Bildhauer and Costal, et. al [14, 31]. After a comprehensive analysis of these notions Bildhauer concludes [14]:

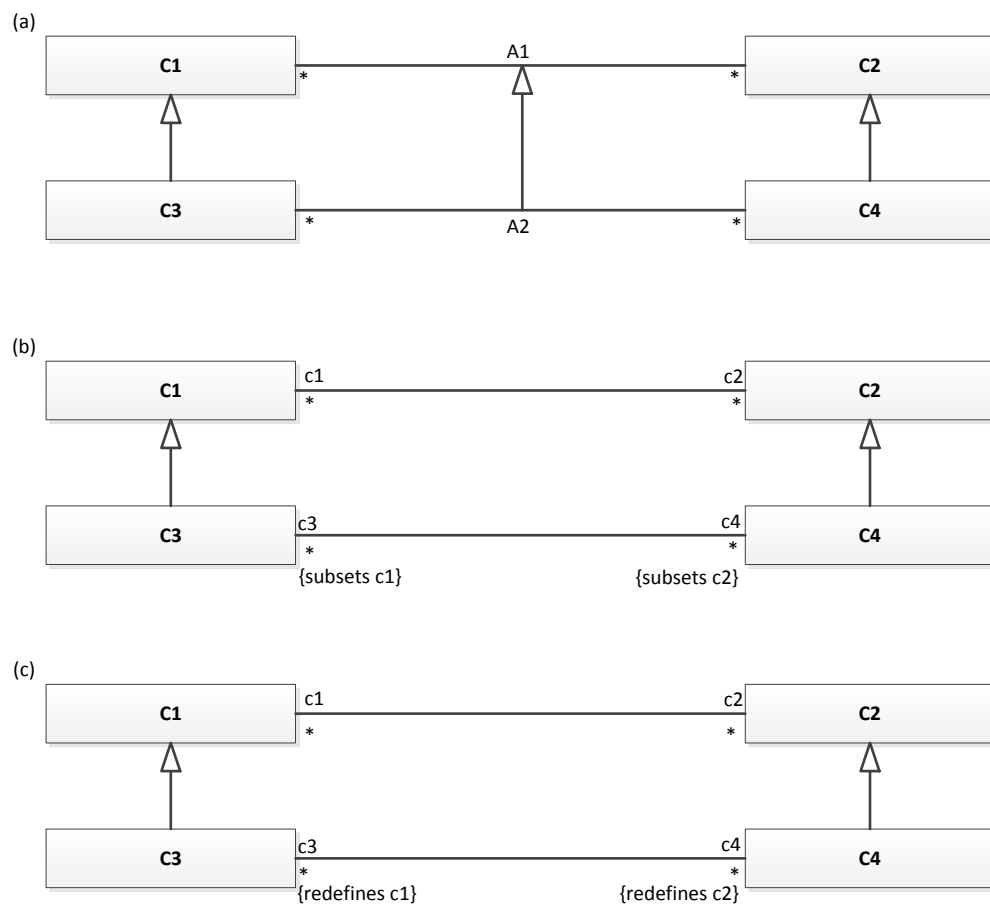


Figure 2.7: Class diagrams illustrating association specialization, subsetting and redefinition.

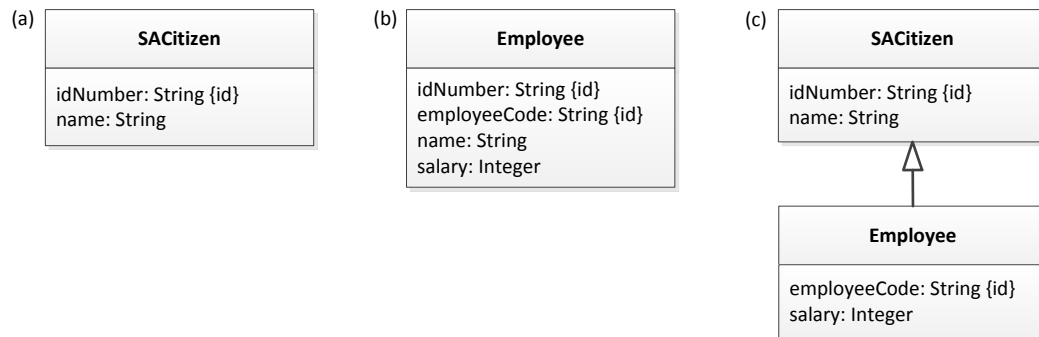


Figure 2.8: Examples of identity constraints applied to classes.

“The specialization of an association is equivalent to the subsetting of (one of) its ends and redefinition can be treated as a subconcept of subsetting.”

Indeed, the UML specification states explicitly that subsetting or redefinition of association ends respectively implies association specialization of the relevant association [64]. Therefore, in this dissertation no further attempt will be made to explicitly distinguish the notions of association specialization, subsetting and redefinition. Rather, the simplifying assumption will be made that these notions are all a form of association specialization.

2.1.6 Identity Constraints

Version 2.4.1 of the UML specification introduces means via which instances of a class can be identified uniquely. To indicate that an attribute forms part of the identity of a class, an `id` property modifier is applied to it [64]. An example of this is shown in Figure 2.8(a). The `{id}` property modifier next to the attribute `idNumber` indicates that it is not permitted for the class **SACitizen** to have two or more instances with the same value for the `idNumber` attribute.

When multiple attributes are marked on a single class with the `{id}` property modifier, it represents a compound key [64]. Thus, instances of the **Employee** class in Figure 2.8(b) are uniquely identified by the compound key consisting of the attributes `idNumber` and `employeeCode`.

According to the UML specification a compound key can also be represented by applying the `id` property modifier across classes in an inheritance hierarchy as shown in Figure 2.8(c) [64]. That is, instances of the **Employee** class are uniquely identified by the compound key consisting of the attributes `idNumber` (defined in the **SACitizen**) and `employeeCode`. In Section 5.1 (p. 61) it will be shown that this interpretation is problematic.



Figure 2.9: Classes C1 and C2 both have an attribute called a.



Figure 2.10: Classes C1 and C2 both have an operation with name f

2.1.7 Qualified Names

In UML the names of classes, associations, association ends (roles), attributes, operations and parameters to operations are all unique due to the use of qualified names [64]. This section discusses the impact of qualified names on associations/attributes and operations and their parameters.

Consider the case of an attribute **a** appearing in two different classes as is shown in Figure 2.9. The name of attribute **a** is qualified by the classes **C1** and **C2** resulting respectively in the qualified names **C1::a** and **C2::a**. Hence, even though the unqualified name of the attribute **a** may be the same for both the classes **C1** and **C2**, the qualified names of the attribute **a** for classes **C1** and **C2** are different. Thus, an attribute **a** appearing in class **C1** can be distinguished from an attribute **a** appearing in class **C2**.

Figure 2.10 shows operation **f** appearing in the classes **C1** and **C2** with parameters **p1:P1, ..., pm:Pm** and **p1:P1, ..., pn:Pn** respectively. The qualified name of operation **f** for class **C1** is **C1::f** and **C2::f** for class **C2**. Parameter names are qualified similarly. That is, the qualified names of the parameters **p1, ..., pm** for class **C1** are **C1::p1, ..., C1::pm**. Similarly the qualified parameter names of **p1, ..., pn** for class **C2** are **C2::p1, ..., C2::pn**.

2.2 Modelling Heuristics

The aim of this section is by no means to provide a comprehensive treatise on object-oriented modelling heuristics. Rather, this section is constrained to only those heuristics that are meaningful from a conceptual modelling perspective and that can (as is illustrated in Chapter 7 on p. 119) be detected via formal scenario testing. In particular the focus will be on heuristics that support conceptual modelling and therefore heuristics that are concerned with implementation details are intentionally excluded. The objective of the heuristics discussed here is therefore to detect various forms of classification violations. For this purpose notions of class cohesion and inheritance cohesion are considered as is defined by Eder, et. al. [38].

Eder, et. al. distinguish five degrees of class cohesion namely *separable-*, *multifaceted-*, *non-delegated-*, *concealed-* and *model class cohesion*. *Model class cohesion* is the most ideal form of class cohesion. A class has model class cohesion when it does not suffer of separable-, multifaceted-, non-delegated- or concealed class cohesion. Instead the class represents a single semantically meaningful concept. Undesirable degrees of class cohesion are discussed in Sections 2.2.1 - 2.2.4.

Eder, et. al. rate *inheritance cohesion* of an inheritance hierarchy as high if the inheritance hierarchy represents a generalization hierarchy in the sense of conceptual modelling [38]. From a conceptual modelling perspective a subclass of a more general class is understood as representing a proper subset of the general class. That is, instances of the subclass are also instances of the more general class [49, 64]. Low inheritance cohesion occurs where attributes and/or operations are defined in the general class that are not applicable to every subclass. Inheritance cohesion is discussed for attributes and operations in Sections 2.2.5 (p. 27) and 2.2.6 (p. 28) respectively.

2.2.1 Separable Class Cohesion

Separable class cohesion indicates a class that consists of multiple unrelated semantic concepts. Separable class cohesion is exemplified by a class that consists of one or more disjoint groups of attributes and/or operations where each group is used in isolation of other groups.

As an example, consider the `Employee` class in Figure 2.11. The `employeeCode`, `employeeName` and `salary` attributes along with the `calculateSalaryIncrease(increase:Integer)` operation only pertain to employees, while the `projectName`, `projectCost1`, `projectCost2` attributes and the `calculateProjectCost()` operation relate to project concerns.

To make the difference in use for the employee and project concepts explicit, example instances of the `Employee` class are provided in Figure 2.12. It illustrates an UML object diagram with instances called `pablo` and `projectX`. It is clear that the `calculateProjectCost()` operation is not meaningful for the `pablo` instance. The `calculateSalaryIncrease(increase:Integer)` operation is similarly not sensible for the `projectX` instance. In order to address separable class cohesion it is shown in Figure 2.13 how the employee and project concepts can be split into separate classes. Figure 2.14 illustrates that creating instances of the classes `Employee` and `Project` are now more meaningful when the semantic concepts are separated.

2.2.2 Multifaceted Class Cohesion

A class has *multifaceted class cohesion* if it is not separable and if, when the set of attributes of the class is interpreted as a relation schema, it is found to not be in second normal form (2NF). Here 2NF is simply defined to mean that each non-key attribute is functionally dependent on the complete key. For an in-depth discussion on the nuances of normalization the reader is

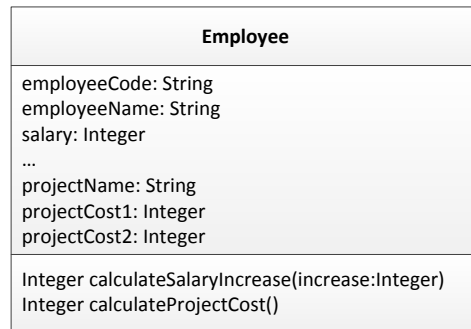


Figure 2.11: The `Employee` class has separable class cohesion.



Figure 2.12: `pablo` and `projectX` are both instances of the `Employee` class.

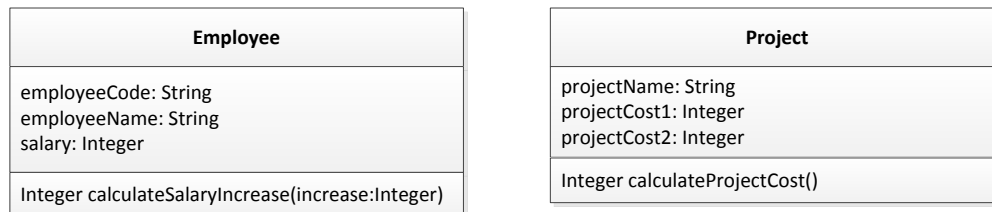


Figure 2.13: Splitting the `Employee` class into `Employee` and `Project` classes.

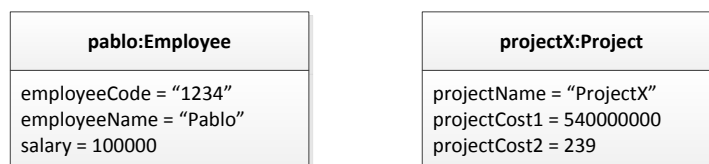


Figure 2.14: Setting project related properties on the `Project` class.

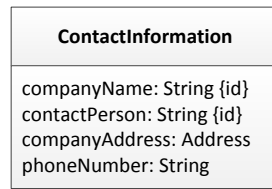


Figure 2.15: The `ContactInformation` class has multifaceted class cohesion.

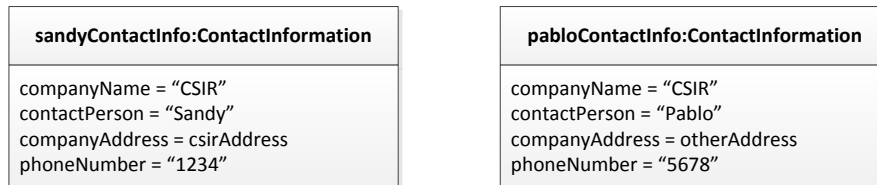


Figure 2.16: Instances of the `ContactInformation` class.

referred to the references provided by Eder, et. al. [38] in this regard (see for instance [32, 55]).

The class `ContactInformation` in Figure 2.15 represents the contact information of a person who serves as a contact person of a particular company. It is possible that the same company can have more than one contact person, each with their own phone number. Thus `companyName` and `contactPerson` serve as the compound key of the `ContactInformation` class. Furthermore, the assumption is made that a single address is associated with a company. Therefore, the `ContactInformation` class is not in 2NF because the `companyAddress` attribute is only dependent on the `companyName` part of the key and not the complete key. Since the `ContactInformation` class is not in 2NF, it will be possible to create two different instances of `ContactInformation` such that the instances have the same `companyName` values but different values for the `companyAddress` attribute. This is illustrated in Figure 2.16.

The problem can be addressed by redesigning the `ContactInformation` class by introducing a `Company` class, which holds the `companyName` and `companyAddress` attributes where `companyName` serves as the key of the `Company` class. In the `ContactInformation` class the `companyName` and `companyAddress` attributes are replaced with a `company` attribute. This is shown in Figure 2.17 and in Figure 2.18 it is shown how the instances of Figure 2.16 can be represented in the redesign.

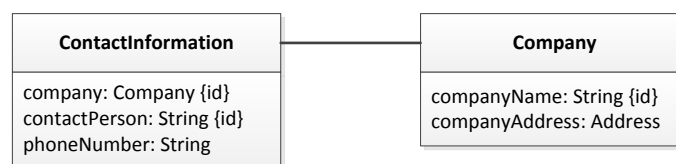
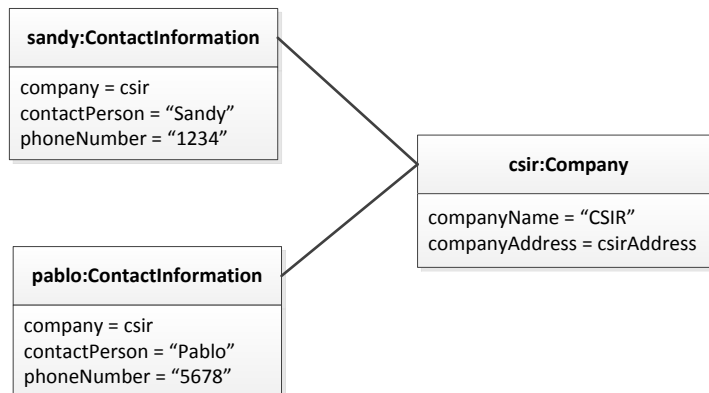
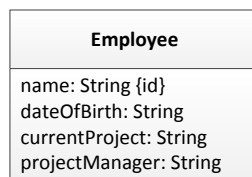


Figure 2.17: The redesign of `ContactInformation` class.

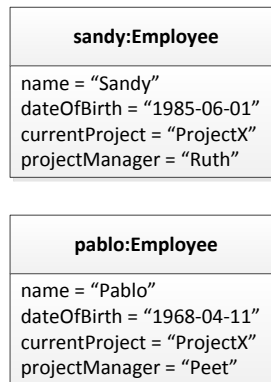
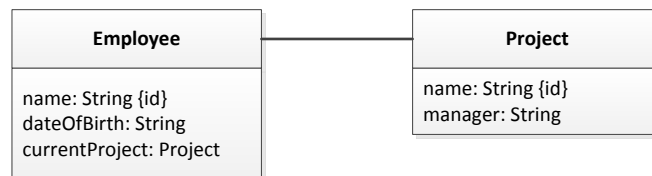
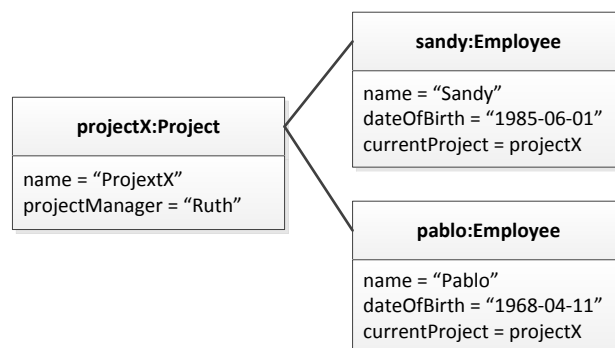
Figure 2.18: Instances of the redesigned `ContactInformation` class.Figure 2.19: The `Employee` class has non-delegated class cohesion.

2.2.3 Non-delegated Class Cohesion

If the set of attributes of a class is interpreted as a relation schema and they are found to not be in third normal form (3NF), the class cohesion of the class is said to be *non-delegated* if its cohesion is neither separable nor multifaceted. A schema relation is in 3NF if it is in 2NF and it does not contain transitive dependencies.

Consider the `Employee` class in Figure 2.19. The attributes `dateOfBirth` and `currentProject` are functionally dependent on the `name` attribute. If the assumption is made that a given project always has one project manager, there is also a functional dependency between the `currentProject` and `projectManager` attributes. This results in there being a transitive dependency between `name` and `projectManager`. Hence, the `Employee` class is not in 3NF. It is therefore possible to have two instances of the `Employee` class such that they have the same value for the attribute `currentProject`, but not for the attribute `projectManager`. This is illustrated in Figure 2.20.

This semantic anomaly can be avoided by extracting the project concept into a separate class. This redesign is illustrated in Figure 2.21 and in Figure 2.22 it is shown that with the redesigned classes it is impossible to have different project managers for the same project.

Figure 2.20: Instances of the `Employee` class.Figure 2.21: Extracting the project concept into its own `Project` class.Figure 2.22: Instances of the redesigned `Employee` and `Project` classes.

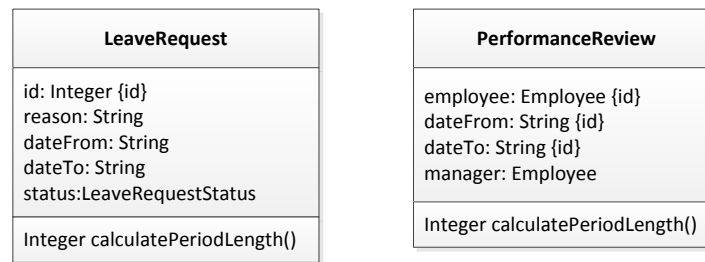


Figure 2.23: The classes `LeaveRequest` and `PerformanceReview` have concealed class cohesion.

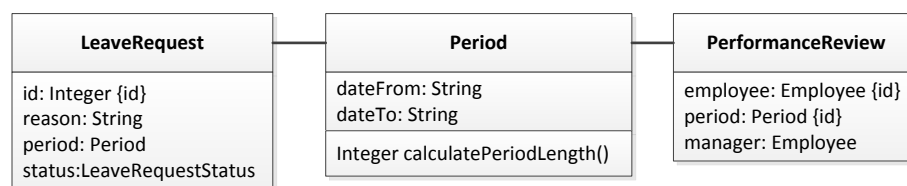


Figure 2.24: Redesign of the `LeaveRequest` and `PerformanceReview` with `Period` class.

2.2.4 Concealed Class Cohesion

A class has *concealed class cohesion* when it does not have separable-, multifaceted- or non-delegated class cohesion, but there is still a useful concept concealed within the class. An example of this can be seen in the `LeaveRequest` class presented in Figure 2.23. Even though the `fromDate` and `toDate` attributes along with the `calculatePeriodLength()` operation are related and relevant to the `LeaveRequest` class, this combination of attributes and operation represents a concealed concept.

Concealed class cohesion becomes even more apparent where two or more classes exist, which share the same subset of attributes and/or operations. As can be seen in Figure 2.23 the `PerformanceReview` class shares the `fromDate` and `toDate` attributes and `calculatePeriodLength()` operation with the `LeaveRequest` class. This is an indication of concealed class cohesion. The redesigned `LeaveRequest` and `PerformanceReview` classes are illustrated in Figure 2.24.

2.2.5 Low Inheritance Cohesion of Attributes

In Figure 2.25 an example, which is often mentioned in the literature [87, 89], is illustrated of low inheritance cohesion of attributes. Since the `Rectangle` class provides both a `width` and a `height` attribute, the `Square` class (since it specializes the `Rectangle` class) is forced to ignore one of the attributes of the `Rectangle` class in order to stay true to the semantics of a square.

This design can be improved by having both the `Rectangle` and `Square` classes specialize

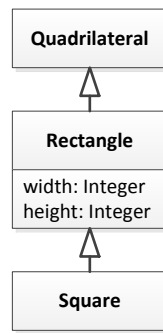


Figure 2.25: The **Square** and **Rectangle** classes have low inheritance cohesion.

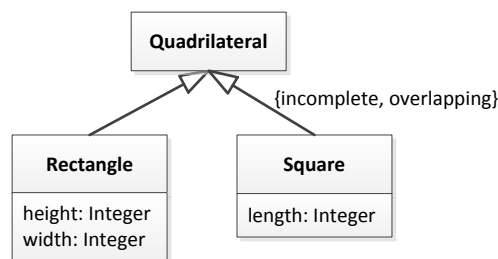


Figure 2.26: A redesign of the **Square** and **Rectangle** classes.

the **Quadrilateral** class, where the **Square** class is defined as having only a `length` attribute and the **Rectangle** class is defined to have the attributes `height` and `width`. The annotation `{incomplete, overlapping}` is added to emphasize that other quadrilaterals may exist and that rectangles and squares may overlap when the width and height of a rectangle are equal. This redesign is illustrated in Figure 2.26.

2.2.6 Low Inheritance Cohesion of Operations

Operations form an essential part of classification in UML class diagrams. Low inheritance cohesion with regards to operations is best expressed in terms of the Liskov substitution principle. The Liskov substitution principle states that a subclass must be substitutable by the more general class [83, 87]. Consider the **Bird** and **Penguin** classes of Figure 2.27. If an instance of the **Penguin** class is used as if it is an instance of the **Bird** class, the instance of the **Penguin** class will be allowed to `fly()` and `walk()`. It makes sense for a penguin to be able to walk but not to fly. Allowing a penguin to fly is an example of a Liskov substitution principle violation, while allowing a penguin to walk agrees with the Liskov substitution principle.

In Figure 2.28 the Liskov substitution principle violation is addressed by adding the **FlyingBird** and **FlightlessBird** classes as direct subclasses of the **Bird** class. Moving the `fly()` operation from the **Bird** class to the **FlyingBird** class ensures that only birds that can fly are able to fly. The **Penguin** class is now defined as a subclass of the **FlightlessBird**

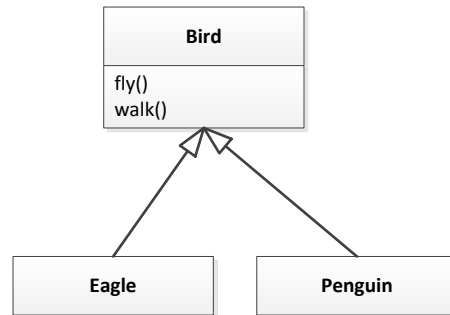


Figure 2.27: The Bird class hierarchy have low inheritance cohesion.

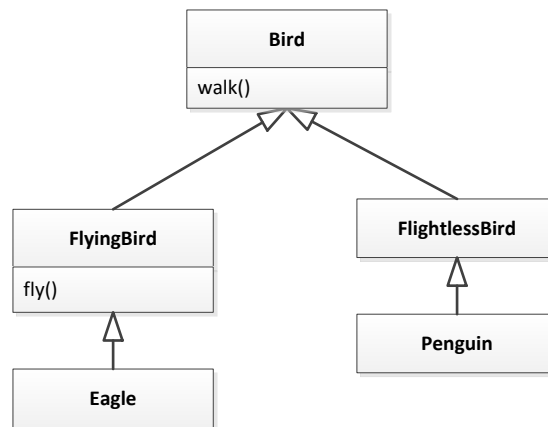


Figure 2.28: A redesign of the Bird inheritance hierarchy.

class, while the `Eagle` class is defined as a subclass of `FlyingBird` class. By addressing the Liskov substitution principle violation, inheritance cohesion of the `Bird` inheritance hierarchy is improved.

2.3 Summary

This chapter provided a brief introduction into UML class diagrams and related modelling heuristics that will be used for formal scenario testing in Chapters 6 (p. 92) and 7 (p. 119). This chapter discussed classes and data types with their attributes and operations. Binary associations was discussed and it was made explicit that the end of a binary association corresponds with an attribute. The discussion on the essential elements of the UML class diagram notation concluded with a discussion on the specialization of classes. The discussion on advanced UML class diagram notation included association specialization and identity constraints.

Modelling heuristics for detecting low class cohesion (separable-, multifaceted-, non-delegated- and concealed class cohesion) and low inheritance cohesion were discussed. For all occurrences of low cohesion the means to improve cohesion have been discussed. In Chapter 7 (p. 119) it will be shown how violations of these heuristics can be detected using formal scenario testing.

Chapter 3

Description Logics and OWL 2

This chapter gives a general introduction to DLs (Section 3.1), which is followed by descriptions for the DLs \mathcal{ALCQI} and $\mathcal{SROIQ}^{(D)}$, as well as OWL 2, in Section 3.2 (p. 40). Relevant reasoning tasks are discussed in Section 3.3 (p. 46).

3.1 Description Logics

Description logics (DLs) are often described as syntactic variants of first-order logic [19, 105] that are specifically designed for the conceptual representation of an application domain in terms of concepts and relationships between concepts [8, 6, 13, 24, 78]. The name *Description Logics* was chosen to emphasise that this family of knowledge representation formalisms is used to describe a domain of interest in terms of *concept descriptions*, and on the other hand that these formalisms have a *formal logic-based* semantics [6]. An introduction to DLs is given in Section 3.1.1 and the formal logic-based semantics of DLs are discussed in Section 3.1.2 (p. 33). In Section 3.1.3 (p. 33) the syntax and semantics of DLs are elucidated by providing the syntax and semantics of the DL \mathcal{AL} along with examples of concept descriptions that are either permissible or unacceptable in \mathcal{AL} . The extension of DLs with data types are discussed in Section 3.1.4 (p. 36). The expressivity of a DL depends on the constructors that are defined for the DL. This is discussed in Section 3.1.5 (p. 38).

3.1.1 Description Logic Primer

The basic syntactic building blocks for an arbitrary DL \mathcal{L} are based on the three disjoint sets N_C , N_R and N_I , where N_C is a set of concept names, N_R is a set of role names and N_I is a set of individual names. Concept names represent classes of entities (called concepts) that share common characteristics, roles names denote binary relations (called roles) that exist between individuals and individual names are used to refer to specific instances (called individuals) in a domain of interest [8, 94, 76, 103].

Ontologies (or knowledge bases) constructed using DLs can consist of three possible building blocks namely, the TBox, the ABox and the RBox. The TBox is used to define the vocabulary or terminology of the application domain along with the relations that exist

between concepts. Axioms in the TBox have the form $C_1 \sqsubseteq C_2$ (C_2 subsumes C_1 or C_1 is subsumed by C_2) or $C_1 \equiv C_2$ (C_1 is equivalent to C_2) where C_1 and C_2 are concepts [8, 94, 103]. As an example, it is possible to state that a woman is a kind of person using the axiom

$$Woman \sqsubseteq Person$$

Roles can be used to define relations between concepts. As an example,

$$Parent \equiv \exists hasChild.Person$$

states that every parent has at least one child whom is a person.

While the TBox defines general knowledge about the application domain, the ABox is used to assert specific knowledge regarding particular individuals in the application domain. For C a concept, r a role, and a and b individuals, ABox assertions are of the form [8, 94, 103]

- $C(a)$, which states that the individual a is an instance of the concept C ,
- $r(a, b)$, which asserts that the individuals a and b are related by r ,
- $\neg r(a, b)$, which asserts that the individuals a and b are not related by r ,
- $a \approx b$, which states that the individual names a and b refer to the same individual and
- $a \not\approx b$, which states that the individual names a and b refer to different individuals.

As an example, consider the concept *Woman* and the individual *mary*. Then the assertion

$$Woman(mary)$$

states that Mary is a woman. The assertion

$$hasChild(mary, susan)$$

states that the individual *mary* has a child, which is the individual *susan* using the *hasChild* role.

Depending on the expressivity of the DL used to construct the ontology, the ontology may include an RBox. An RBox is used to define relations between roles, which consists of role inclusion axioms and role property assertions. Role inclusion axioms are of the form $r \sqsubseteq s$ where r and s are roles. Role property assertions are used to assert role properties like reflexivity, transitivity, symmetry and disjointness [58, 103]. As an example, the role inclusion axiom

$$hasChild \sqsubseteq hasFamily$$

states that the *hasChild* binary relation is a kind of *hasFamily* binary relation. The role property assertion

$$\text{Dis}(\text{hasParent}, \text{hasChild})$$

states that a given individual cannot have a single individual as both parent and child.

Throughout this chapter A , C and D denote concepts, r , s and t denote roles and a and b denote individuals.

3.1.2 Semantics

The formal logic-based semantics of DLs are defined in a model-theoretic way. That is, the semantics of a DL are defined by assigning an interpretation to the concept constructors and (depending on the expressivity of the DL) role constructors, that are defined for the DL [8, 6, 94, 103].

Definition 3.1. An *interpretation* for an arbitrary DL \mathcal{L} , denoted by \mathcal{I} , is defined as $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ denotes the domain of interest and $\cdot^{\mathcal{I}}$ denotes a mapping function that maps every concept name $C \in N_C$ in \mathcal{L} to a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, every role name $r \in N_R$ in \mathcal{L} to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and every individual name $a \in N_I$ in \mathcal{L} to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. \diamond

Definition 3.2. An interpretation \mathcal{I} *satisfies* the respective TBox, ABox and RBox axioms based on the conditions defined in Table 3.1. When an interpretation \mathcal{I} satisfies axiom α , it is written as $\mathcal{I} \models \alpha$. An axiom α is called *satisfiable* if and only if there is at least one interpretation \mathcal{I} such that $\mathcal{I} \models \alpha$ [58, 76, 103]. \diamond

The interpretation function is extended to arbitrary concept descriptions by the inductive application of the concept constructors defined for the particular DL \mathcal{L} [8, 94, 103]. The model-theoretic semantics of DLs are given by Definitions 3.3 and 3.4.

Definition 3.3. \mathcal{I} is called a *model* of an axiom α if α is satisfied by the interpretation \mathcal{I} . \mathcal{I} is called a *model* of an ontology \mathcal{O} , written as $\mathcal{I} \models \mathcal{O}$, if for every $\alpha \in \mathcal{O}$ it follows that $\mathcal{I} \models \alpha$ [36, 94, 103]. \diamond

Definition 3.4. An axiom α is said to be *entailed* by an ontology \mathcal{O} , written as $\mathcal{O} \models \alpha$, if every model of \mathcal{O} is also a model of α . Thus, for every $\mathcal{I} \models \mathcal{O}$, $\mathcal{I} \models \alpha$ holds. It is also said that axiom α is the *logical consequence* of ontology \mathcal{O} . [8, 103]. \diamond

3.1.3 \mathcal{AL}

The different DLs are characterised by the concept constructors they permit respectively. As an example, the DL \mathcal{AL} is considered in Definition 3.5 (p. 35). A summary is given of the \mathcal{AL} concept constructor names and their syntax in Table 3.2 (p. 35).

Table 3.1: Semantics of TBox, ABox and RBox axioms

Atom	Syntax	Semantics
Vocabulary		
Atomic concept N_C	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Atomic role N_R	p	$p^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Individual N_I	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Axiom name	Axiom α	Condition for $I \models \alpha$
TBox axioms		
Concept inclusion	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Concept equivalence	$C_1 \equiv C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
ABox axioms		
Concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
Role assertion	$r(a, b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$
Negated role assertion	$\neg r(a, b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin r^{\mathcal{I}}$
Individual equality	$a \approx b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
Individual inequality	$a \not\approx b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$
RBox axioms		
Role inclusion	$r_1 \sqsubseteq r_2$	$r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$
Role equivalence	$r_1 \equiv r_2$	$r_1^{\mathcal{I}} = r_2^{\mathcal{I}}$
Complex role inclusion	$r_1 \circ \dots \circ r_n \sqsubseteq r$	$r_1^{\mathcal{I}} \circ \dots \circ r_n^{\mathcal{I}} \subseteq r^{\mathcal{I}}$
Role transitivity	$\text{Tra}(r)$	$r^{\mathcal{I}} \circ r^{\mathcal{I}} \subseteq r^{\mathcal{I}}$
Role reflexivity	$\text{Ref}(r)$	$(x, x) \in r^{\mathcal{I}}$ for all $x \in \Delta^{\mathcal{I}}$
Role irreflexivity	$\text{Irr}(r)$	$(x, x) \notin r^{\mathcal{I}}$ for all $x \in \Delta^{\mathcal{I}}$
Role disjointness	$\text{Dis}(r_1, r_2)$	If $(x, y) \in r_1^{\mathcal{I}}$ then $(x, y) \notin r_2^{\mathcal{I}}$ for all $x, y \in \Delta^{\mathcal{I}}$
Role symmetry	$\text{Sym}(r)$	If $(x, y) \in r^{\mathcal{I}}$ then $(y, x) \in r^{\mathcal{I}}$ for all $x, y \in \Delta^{\mathcal{I}}$
Role asymmetry	$\text{Asy}(r)$	If $(x, y) \in r^{\mathcal{I}}$ then $(y, x) \notin r^{\mathcal{I}}$ for all $x, y \in \Delta^{\mathcal{I}}$
Functional role restriction	$\text{Func}(r)$	$\top \sqsubseteq \leq 1r. \top$
\circ on the right-hand side denotes standard composition of binary relations: $r_1^{\mathcal{I}} \circ r_2^{\mathcal{I}} := \{(x, z) \mid (x, y) \in r_1^{\mathcal{I}}, (y, z) \in r_2^{\mathcal{I}}\}$		

Table 3.2: \mathcal{AL} concept constructors and syntax

Concept constructor	Syntax
Atomic concept	A
Top concept	\top
Bottom concept	\perp
Atomic concept negation	$\neg A$
Concept conjunction	$C_1 \sqcap C_2$
Limited existential restriction	$\exists r. \top$
Universal restriction	$\forall r. C$

Definition 3.5. \mathcal{AL} concept descriptions are defined as follows:

$$C ::= \top \mid \perp \mid A \mid \neg A \mid C_1 \sqcap C_2 \mid \forall r. C \mid \exists r. \top$$

where A is an atomic concept, C_1 and C_2 are (possibly complex) concepts and r a role¹. The semantics of \mathcal{AL} concept descriptions are defined by inductively extending the concept descriptions according to the semantics of the \mathcal{AL} concept constructors as given in Table 3.3 [8].

◇

Assuming the atomic concepts *Person* and *Female* are defined for an \mathcal{AL} TBox. Then, using the conjunction concept constructor (see Table 3.2), it is possible to define the complex concept *Woman* as

$$Woman \equiv Person \sqcap Female \tag{3.1}$$

Furthermore, assume that the atomic role *hasChild* is used to describe the binary relation between individuals that are in a parent-child relation. The complex concept *Mother* can then be defined using the conjunction, limited existential restriction and universal restriction concept constructors (see Table 3.2) as

$$Mother \equiv Woman \sqcap \exists hasChild. \top \sqcap \forall hasChild. Person \tag{3.2}$$

This complex expression (3.2) states that a mother is a woman that has at least one child ($\exists hasChild. \top$) and all her children are persons ($\forall hasChild. Person$). The axiom $\forall hasChild. Person$ is required to ensure only children that are persons (and not pets for example) are considered.

Note that since \mathcal{AL} only includes the limited existential restriction concept constructor rather than the full existential restriction concept constructor, it is not possible to define

¹The \top concept has to be distinguished from the capital letter T.

Table 3.3: \mathcal{AL} concept constructors and semantics

Concept constructor	Semantics
$\top^{\mathcal{I}}$	$\Delta^{\mathcal{I}}$
$\perp^{\mathcal{I}}$	\emptyset
$(\neg A)^{\mathcal{I}}$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
$(C_1 \sqcap C_2)^{\mathcal{I}}$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
$(\forall r.C)^{\mathcal{I}}$	$\{x \in \Delta^{\mathcal{I}} \mid \text{For all } y \in \Delta^{\mathcal{I}}, \text{ if } (x, y) \in r^{\mathcal{I}}, \text{ then } y \in C^{\mathcal{I}}\}$
$(\exists r.C)^{\mathcal{I}}$	$\{x \in \Delta^{\mathcal{I}} \mid \text{There exists an } y \in \Delta^{\mathcal{I}}, \text{ such that } (x, y) \in r^{\mathcal{I}}\}$

Mother as

$$\textit{Mother} \equiv \textit{Woman} \sqcap \exists \textit{hasChild}.\textit{Person} \quad (3.3)$$

The full existential restriction concept constructor is, for example, defined for the DL \mathcal{ALC} , which has the syntax $\exists r.C$ with the semantics defined as [107]

$$(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{There exists an } y \in \Delta^{\mathcal{I}}, \text{ such that } (x, y) \in r^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$$

3.1.4 Extending DLs with Data Types

Extending DLs with data types is based on the idea of data type mapping. Let \mathcal{D} be a set of data types. The function $\cdot^{\mathcal{D}}$ associates for each $d \in \mathcal{D}$, a set $d^{\mathcal{D}} \subseteq \Delta^{\mathcal{D}}$ where $\Delta^{\mathcal{D}}$ is the domain of all data types. A detailed account of data type mappings can be found in the papers of Horrocks, et. al. [60, 59, 90].

Definition 3.6. For a DL \mathcal{L} extended with data types the set of role names N_R are defined as

$$N_R = R_A \cup \{r^- \mid r \in R_A\} \cup R_D$$

where R_D is the set of *concrete role names* and R_A is the set of *abstract role names*. \diamond

Definition 3.7. An interpretation for a DL \mathcal{L} extended with data types is a triple $\mathcal{I} = (\Delta^{\mathcal{I}}, \Delta^{\mathcal{D}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ and $\Delta^{\mathcal{D}}$ are nonempty disjoint sets such that $\Delta^{\mathcal{I}}$ is the domain of interest, $d^{\mathcal{D}} \subseteq \Delta^{\mathcal{D}}$ for each $d \in \mathcal{D}$ and $\cdot^{\mathcal{I}}$ is a function assigning to each concept $C \in N_C$, each abstract role $r \in R_A$, each individual $a \in N_I$, and to each concrete role $t \in R_D$ of \mathcal{L} , interpretations $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and $t^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$ respectively. The functions $\cdot^{\mathcal{D}}$ and $\cdot^{\mathcal{I}}$ are extended to data type concepts as shown in Table 3.4. \diamond

The interpretation \mathcal{I} is a model for an axiom α and a model for an ontology \mathcal{O} as per Definition 3.3 (p. 33).

Table 3.4: Model-theoretic semantics of DLs extended with data types

Vocabulary		
Atom	Syntax	Semantics
Data type d	d	$d^{\mathcal{D}} \subseteq \Delta^{\mathcal{D}}$
Concrete role R_D	t	$t^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$
Data value	v	$v^{\mathcal{I}} = v^{\mathcal{D}}$
Data type axioms		
Axiom name	Axiom α	Condition for $\mathcal{I} \models \alpha$
Concrete role disjointness	$\text{Dis}(t_1, t_2)$	If $(x, y) \in t_1^{\mathcal{I}}$, then $(x, y) \notin t_2^{\mathcal{I}}$, for all $x \in \Delta^{\mathcal{I}}$ and $y \in \Delta^{\mathcal{D}}$
Concrete role inclusion	$t_1 \sqsubseteq t_2$	$t_1^{\mathcal{I}} \subseteq t_2^{\mathcal{I}}$
Concrete role assertion	$t(a, v)$	$(a^{\mathcal{I}}, v^{\mathcal{D}}) \in t^{\mathcal{I}}$
Concept constructors using data types		
Constructor name	Syntax	Semantics
Data type at least	$\geq nt.d$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \in \Delta^{\mathcal{D}} \mid (x, y) \in t^{\mathcal{I}} \text{ and } y \in d^{\mathcal{D}}\} \geq n\}$
Data type at most restriction	$\leq nt.d$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \in \Delta^{\mathcal{D}} \mid (x, y) \in t^{\mathcal{I}} \text{ and } y \in d^{\mathcal{D}}\} \leq n\}$
Data type universal restriction	$\forall t.d$	$\{x \in \Delta^{\mathcal{I}} \mid \text{For all } y \in \Delta^{\mathcal{D}}, \text{ if } (x, y) \in t^{\mathcal{I}}, \text{ then } y \in d^{\mathcal{D}}\}$
Data type existential restriction	$\exists t.d$	$\{x \in \Delta^{\mathcal{I}} \mid \text{An } y \in \Delta^{\mathcal{D}} \text{ exists such that } (x, y) \in t^{\mathcal{I}} \text{ and } y \in d^{\mathcal{D}}\}$

Table 3.5: Nomenclature for important DL features

Symbol	Expressive feature	Example
\mathcal{C}	Complex concept negation	$\neg C$
\mathcal{U}	Concept disjunction	$C_1 \sqcup C_2$
\mathcal{E}	Full existential restriction	$\exists r.C$
\mathcal{I}	Inverse roles	r^-
\mathcal{Q}	Qualified number restrictions	$\leq 3r.C$
\mathcal{N}	Unqualified number restrictions	$\leq 3r.\top$
\mathcal{F}	Functional role restriction	$\top \sqsubseteq \leq 1r.\top$ or $\text{Func}(r)$
\mathcal{O}	Nominals	$\{a\}$
\mathcal{H}	Role hierarchies	$r_1 \sqsubseteq r_2$
\mathcal{R}	Limited complex role inclusion	$r_1 \circ r_2 \sqsubseteq r$
\mathcal{S}	Abbreviation for \mathcal{ACC} with transitive roles	$r \circ r \sqsubseteq r$
\mathcal{D}	Concrete data types	$\exists t.\text{String}$ or $\leq 10t.\text{Integer}$

3.1.5 DL Nomenclature

The main goal in designing DLs is to achieve and maintain desirable computational complexity properties for core reasoning tasks while retaining expressive power. As such, for many DLs the core reasoning tasks are decidable and well-behaved in practice. This is achieved by limiting the expressivity of the DL in different ways. However, for a DL to be useful in an application domain, it needs to be expressive enough to describe the domain accurately. Since DLs are used in a wide variety of application domains with conflicting expressivity demands, it gave rise to the design of different DLs with different levels of expressivity and computational complexity. Consequently, it is possible that a given DL may be suitable for one application domain, while it may be impractical for another [6, 76, 103].

More expressive DLs are designed by extending an inexpressive DL like \mathcal{AL} . As an example, the DL \mathcal{ACC} extends the DL \mathcal{AL} with complex concept negation, full existential restriction and concept disjunction [8, 60, 76, 103].

In order to discern the expressivity of different DLs, an informal naming convention has been established, which gives some hint as to the expressivity of the DL. Table 3.5 provides an abbreviated list of notations representing different DL features [8, 60, 59, 76, 56, 103].

3.1.6 Characteristics of DLs

In this section, characteristics that are core to DL semantics are discussed. These are the open-world assumption, monotonicity and the unique name assumption [103].

Open-world versus Closed-world Assumption

DLs have been designed with the explicit intention to be able to deal with incomplete information. Consequently, DLs intentionally do not make any assumptions with regards with information that is not known. In particular, no assumption is made about the truth or falsehood of facts that cannot be deduced from the ontology. This approach is known as the *open-world assumption* (OWA). This approach is in contrast with the *closed-world assumption* (CWA) typically used in information systems. With the CWA facts that cannot be deduced from a knowledge base (i.e. database) are implicitly understood as being false [8, 77, 103].

As an example, consider a knowledge base (ontology or database), which contains the single fact $hasChild(mary, susan)$. Thus, it contains only one fact which states that Susan is the child of Mary. From a DL perspective the only deduction that can be made is that Mary has a child called Susan. From the OWA it follows that it is not known whether Mary has other children. Indeed, it is possible for the existence of (a potentially infinite number) of different interpretations, in which Mary either has only one child, only two children or any number of children greater than one. Each of these interpretations represents a model in which the fact $hasChild(mary, susan)$ is true. On the other hand, when a database containing this single fact is considered, the implicit consequence is that Mary has one child only, called Susan. Moreover, from a database perspective, only a single world (or model) is considered where Mary has one child, and one child only.

Monotonic Semantics

Definition 3.4 (p. 33) states that, if an axiom α holds in all models of an ontology \mathcal{O} , α is a logical consequence of \mathcal{O} . As more axioms are added to an ontology, the fewer models the ontology has and the more logical consequences follow from the ontology. Hence, the semantics of DLs are monotonic. In particular, adding new axioms to an ontology does not cause existing logical consequences to be invalidated [78].

Unique Name Assumption

In general, DLs do not make the *unique name assumption* (UNA): different individual names may denote the same individual. To make explicit that different individual names denote different individuals, an individual inequality assertion has to be applied [8, 103]. As an example, to make explicit that the individuals names a and b denote different individuals, the assertion $a \neq b$ has to be added to the ontology.

Table 3.6: Syntax and semantics of \mathcal{ALCQI}

Constructor name	Syntax	Semantics
Atomic concept	A	$A^{\mathcal{I}}$
Concept negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Concept conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
At most restriction	$\leq nr.C$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \in \Delta^{\mathcal{I}} \mid (x, y) \in r^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leq n\}$
Inverse role	r^{-}	$\{(x, y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (y, x) \in r^{\mathcal{I}}\}$

3.2 DLs for the Translation of UML Class Diagrams

The purpose of this section is to introduce the DLs \mathcal{ALCQI} (Section 3.2.1) and $\mathcal{SROIQ}^{(\mathcal{D})}$ (Section 3.2.2) as well as OWL 2, the Web Ontology Language (Section 3.2.3). \mathcal{ALCQI} is a strict subset of $\mathcal{SROIQ}^{(\mathcal{D})}$, with $\mathcal{SROIQ}^{(\mathcal{D})}$ forming the basis for the formal semantics of OWL 2.

UML class diagrams have been translated by Berardi, et. al. to \mathcal{ALCQI} [13] and by Zedlitz, et. al. to OWL 2 [118]. This is discussed in Chapter 4 (p. 49). Refinements to the translation of UML class diagrams to $\mathcal{SROIQ}^{(\mathcal{D})}$ and OWL 2 (for the purpose of formal scenario testing) are discussed in Chapter 5 (p. 60).

3.2.1 \mathcal{ALCQI}

Definition 3.8. Concept expressions for the DL \mathcal{ALCQI} are defined by the grammar

$$C ::= A \mid \neg C \mid C_1 \sqcap C_2 \mid \leq nr.C$$

where n is a non-negative integer, A is an atomic concept, C, C_1, C_2 are (possibly complex) concepts and r is a role. Role expressions for \mathcal{ALCQI} are defined by

$$r ::= p \mid p^{-}$$

where p is an atomic role and p^{-} is the inverse of atomic role p . The semantics of \mathcal{ALCQI} concept and role constructors are given in Table 3.6. From the \mathcal{ALCQI} concept constructors further useful concept expressions can be derived, which are abbreviated in Table 3.7 [13, 85]. \diamond

The abbreviations listed in Table 3.7 are used in the translation of UML class diagrams to \mathcal{ALCQI} by Berardi, et. al. [13].

Table 3.7: \mathcal{ALCQI} abbreviations

Abbreviation	Concept expression abbreviated
\top	$A \sqcup \neg A$
\perp	$\neg \top$
$C_1 \sqcup C_2$	$\neg(\neg C_1 \sqcap \neg C_2)$
$C_1 \Rightarrow C_2$	$\neg C_1 \sqcup C_2$
$\geq nr.C$	$\neg(\leq n - 1r.C)$
$\exists r.C$	$\geq 1r.C$
$\forall r.C$	$\neg \exists r.\neg C$

3.2.2 $\mathcal{SROIQ}^{(D)}$

In this section the syntax and semantics of the DL $\mathcal{SROIQ}^{(D)}$ are explained. However, before proceeding, some prerequisite notions are defined.

Definition 3.9. The set R^s of *simple roles* is defined as

$$R^s = R_A \setminus R^n$$

where R^n , the set of *non-simple roles*, is defined as [58, 103]:

- Every role r occurring in $r_1 \circ \dots \circ r_n \sqsubseteq r$, where $n > 1$, is non-simple.
- Every role r occurring in a simple role inclusion $r_1 \sqsubseteq r$ with a non-simple role r_1 is itself non-simple.
- If r is non-simple, so is $Inv(r)$ where the function Inv is defined such that $Inv(r^-) = r$ and $Inv(r) = r^-$.
- No other role is non-simple.

◇

Definition 3.10. A role hierarchy is *regular* if there is a strict partial ordering \prec on non-simple roles R^n such that

- $r_1 \prec r$ if and only if $Inv(r_1) \prec r$, and
- every role inclusion axiom is of one of the forms

$$r \circ r \sqsubseteq r,$$

$$Inv(r) \sqsubseteq r,$$

$$r_1 \circ \dots \circ r_n \sqsubseteq r,$$

Table 3.8: Syntax and semantics of $SR\mathcal{OIQ}^{(D)}$

Vocabulary		
Atom	Syntax	Semantics
Atomic concept N_C	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Simple role R^s	s	$s^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, adhering to the constraints of Definition 3.9
Non-simple role R^n	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Concrete role R_D	t	$t^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$
Individual N_I	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Data type d	d	$d^{\mathcal{D}} \subseteq \Delta^{\mathcal{D}}$
Data value	v	$v^{\mathcal{I}} = v^{\mathcal{D}}$
Constructors		
Constructor name	Syntax	Semantics
Top concept	\top	$\Delta^{\mathcal{I}}$
Bottom concept	\perp	\emptyset
Atomic concept	A	$A^{\mathcal{I}}$
Nominal	$\{o\}$	$\{o^{\mathcal{I}}\}$
Self restriction	$\exists s.\text{Self}$	$\{x \in \Delta^{\mathcal{I}} \mid (x, x) \in s^{\mathcal{I}}\}$
Concept negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Concept conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
Concept disjunction	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
Universal restriction	$\forall r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \text{For all } y \in \Delta^{\mathcal{I}}, \text{ if } (x, y) \in r^{\mathcal{I}}, \text{ then } y \in C^{\mathcal{I}}\}$
Existential restriction	$\exists r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \text{There exists an } y \in \Delta^{\mathcal{I}}, \text{ such that } (x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
At least restriction	$\geq ns.C$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \in \Delta^{\mathcal{I}} \mid (x, y) \in s^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geq n\}$
At most restriction	$\leq ns.C$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \in \Delta^{\mathcal{I}} \mid (x, y) \in s^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leq n\}$
Inverse role	r^-	$\{(x, y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (y, x) \in r^{\mathcal{I}}\}$

$$r \circ r_1 \circ \dots \circ r_n \sqsubseteq r,$$

$$r_1 \circ \dots \circ r_n \circ r \sqsubseteq r,$$

such that $r \in R_A$ is a (non-inverse) role name, and $r_i \prec r$ for $i = 1, \dots, n$ whenever r_i is non-simple [58, 103]. \diamond

Definition 3.11. $\mathcal{SROIQ}^{(D)}$ concept descriptions are formed inductively using the following grammar:

$$C ::= \top \mid \perp \mid A \mid \{a\} \mid \exists s. \text{Self} \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \forall r. C \mid \exists r. C \mid \geq ns. C \mid \leq ns. C \mid$$

$$\geq nt. d \mid \leq nt. d \mid \forall t. d \mid \exists t. d$$

where A is an atomic concept, C, C_1, C_2 are concepts, a is an individual, r is an abstract role (see Definition 3.6 on p. 36), s is a simple abstract role, t is a concrete role, d is a data type and n is an integer. The syntax and semantics of $\mathcal{SROIQ}^{(D)}$ concept and role constructors are provided in Table 3.8 (p. 42). The syntax and semantics for concept constructors using data types are as defined in Table 3.4 (p. 37) [62, 60, 59, 58, 61, 90, 103].

Decidability is achieved by ensuring that role hierarchies are regular and by restricting the use of abstract roles to only simple roles appearing in

- self restrictions, at most and at least restrictions and
- irreflexivity and disjointness axioms.

\diamond

3.2.3 OWL 2

The syntax of OWL 2 is specified by the World Wide Web Consortium (W3C) [92]. The W3C OWL 2 specification is defined in terms of axioms (or functional-style syntax), which may not be accessible to non-logicians. Therefore, the OWL 2 Manchester syntax has been defined as a means to provide a more compact and user-friendly syntax for OWL 2 [54, 57]. This dissertation makes use of the OWL 2 Manchester syntax.

The semantics of OWL 2 is based on DLs, and in particular $\mathcal{SROIQ}^{(D)}$ [47, 59, 61, 91], extended with a relaxed form of DL-safe rules for the purpose of modelling key constraints [97].

Table 3.9 provides a summary of the translations from OWL 2 Manchester syntax to OWL 2 functional-style syntax to DL syntax. Only syntax that is pertinent to the current dissertation has been included in the summary. For the full syntax and related translations the reader is referred to the relevant literature [57, 59, 61, 91]. The associated DL semantics follow from the semantics provided in Tables 3.1 (p. 34), 3.4 (p. 37) and 3.8 (p. 42).

From the translations in Table 3.9 it follows that OWL classes, object properties and data properties are translated respectively to DL concepts, abstract roles and concrete roles. In

Table 3.9: Manchester to functional-style to DL syntax translation

Vocabulary		
Name	OWL Atom	DL Atom
OWL class name A	A	A
OWL object property name r	r	r
OWL simple object property name s	s	s
OWL data property name t	t	t
OWL data type name d	d	d
OWL individual name a	a	a
OWL data value v	v	v
Translations		
Manchester syntax	Functional-style syntax	$SR\mathcal{OIQ}^{(D)}$ syntax
Class: A	A	A
Class: C1 EquivalentTo: C2	EquivalentClasses(C1 C2)	$C_1 \equiv C_2$
Class: C1 SubClassOf: C2	SubClassOf(C1 C2)	$C_1 \sqsubseteq C_2$
DisjointClasses: C1, C2	DisjointClasses(C1, C2)	$C_1 \sqsubseteq \neg C_2$
ObjectProperty: r	r	r
ObjectProperty: r1 SubPropertyOf: r2	SubObjectPropertyOf(r1 r2)	$r_1 \sqsubseteq r_2$
ObjectProperty: r Domain: C1	ObjectPropertyDomain(r C1)	$\exists r.T \sqsubseteq C_1$
ObjectProperty: r Range: C2	ObjectPropertyRange(r C2)	$\exists r^-.T \sqsubseteq C_2$
ObjectProperty: r1 InverseOf: r2	InverseObjectProperties(r1 r2)	$r_1 \equiv r_2^-$
DataProperty: t	t	t
DataProperty: t1 SubPropertyOf: t2	SubDataPropertyOf(t1 t2)	$t_1 \sqsubseteq t_2$
DataProperty: t Domain: C1	DataPropertyDomain(t C1)	$\exists t.T \sqsubseteq C_1$
DataProperty: t Range: d	DataPropertyRange(t d)	$T \sqsubseteq \forall t.d$
Individual: a Types: C	ClassAssertion(C a)	$C(a)$
Individual: a Facts: r b	ObjectPropertyAssertion(r a b)	$r(a, b)$
Individual: a Facts: t v	DataPropertyAssertion(t a v)	$t(a, v)$
Individual: a DifferentFrom: b	DifferentIndividuals(a b)	$a \not\approx b$
Individual: a SameAs: b	SameIndividual(a b)	$a \approx b$
(†) r some C	ObjectSomeValuesFrom(r C)	$\exists r.C$
(†) r only C	ObjectAllValuesFrom(r C)	$\forall s.C$
(†) s min n C	ObjectMinCardinality(n s C)	$\geq ns.C$
(†) s max n C	ObjectMaxCardinality(n s C)	$\leq ns.C$
(†) The Manchester syntax provided in these instances are usually used in conjunction with EquivalentTo or SubClassOf axioms.	As an example a universal restriction can be used to define class C1 as: Class: C1 EquivalentTo: r only C2	

this dissertation, depending on the context of the discussion, either OWL terminology or DL terminology will be used.

Table 3.9 does not include the syntax and semantics of OWL 2 key constraints. Key constraints are of importance to Chapters 5 (p. 60) and 7 (p. 119) of this dissertation. Key constraints in OWL 2, called Easy Keys, are implemented using a form of DL safe rules and hence cannot be formalized using DLs syntax and semantics. The Manchester syntax for an OWL 2 key constraint is [57, 91]

$$\begin{aligned} \text{Class: } C & \\ \text{hasKey } s_1, \dots, s_n, t_1, \dots, t_m & \end{aligned} \quad (3.4)$$

and the functional-style syntax is

$$\text{HasKey}(C (s_1 \dots s_n) (t_1 \dots t_m)) \quad (3.5)$$

The semantics of Easy Keys is given by Definition 3.12, as is defined by Parsia, et. al. [97].

Definition 3.12. A key constraint is a statement of the form

$$C \text{ hasKey}(s_1, \dots, s_n, t_1, \dots, t_m) \quad (3.6)$$

where s_1, \dots, s_n are simple object properties, t_1, \dots, t_m are data type properties and C is a class description. The semantics for an OWL 2 key constraint (3.7) is then defined for an ontology \mathcal{O} by an interpretation \mathcal{I} such that \mathcal{I} is a model of \mathcal{O} and satisfies (3.7) below

$$\forall xy \left[\exists z_1 \dots z_n v_1 \dots v_m \left[\alpha \wedge \text{HU}(x) \wedge \text{HU}(y) \wedge \bigwedge_{i=1, \dots, n} \text{HU}(z_i) \right] \rightarrow x = y \right], \quad (3.7)$$

where HU holds for all the named individuals in \mathcal{O} (that is, HU is a Herbrand universe) and α is defined as

$$\alpha = C(x) \wedge C(y) \wedge \bigwedge_{i=1, \dots, n} (s_i(x, z_i) \wedge s_i(y, z_i)) \wedge \bigwedge_{i=1, \dots, m} (t_i(x, v_i) \wedge t_i(y, v_i)) \quad (3.8)$$

◇

Equation (3.7) states that the named individuals x and y of type C represent the same individual, if and only if, they agree on

1. their simple object properties s_i with named individuals z_i , for $1 \leq i \leq n$, and
2. their data properties t_i with data values v_i , for $1 \leq i \leq m$, respectively.

Note that Easy Keys is restricted to apply to simple object properties relating named individuals in order to ensure decidability. Moreover, note that object properties and data properties are treated in different ways. In particular, z_1, \dots, z_n represent named individuals while v_1, \dots, v_m are not necessarily named values. This difference in treatment is due to data properties not requiring restrictions in order to achieve decidability and some counter-intuitive inferences that follow if named values are enforced for data types [97].

3.3 Reasoning Tasks

In this section reasoning tasks that are of importance for this dissertation are discussed. DLs are equipped with a number of reasoning procedures, which can be utilized to infer implicit knowledge from the knowledge stated explicitly in the ontology. The pertinent reasoning tasks in this regard are discussed in Section 3.3.1. Reasoning tasks that are focussed on finding and eliminating modelling errors in ontologies are discussed in Section 3.3.2 (p. 47).

3.3.1 Deductive Reasoning Tasks

Axiom satisfiability and axiom entailment have been defined in Definitions 3.2 (p. 33), and 3.4 (p. 33) respectively. Ontology satisfiability (also called ontology consistency), concept satisfiability, subsumption, classification and instance checking are defined next.

Definition 3.13. A concept C is *satisfiable* with respect to an ontology \mathcal{O} if there exists a model \mathcal{I} of \mathcal{O} such that $C^{\mathcal{I}} \neq \emptyset$. A concept C is *unsatisfiable* if and only if for all interpretations \mathcal{I} it follows that $C^{\mathcal{I}} = \emptyset$ [8, 103]. \diamond

Definition 3.14. An axiom α is *satisfiable* with respect to an ontology \mathcal{O} if there exists a model \mathcal{I} of \mathcal{O} such that $\mathcal{I} \models \alpha$. An axiom α is *unsatisfiable* if it does not have a model [36, 94, 103]. \diamond

Satisfiability of roles can be checked by reducing role satisfiability to concept satisfiability. Thus, given a role r , checking the satisfiability of r reduces to checking the satisfiability of the concept $(\geq 1r.\top)$ [66].

Definition 3.15. An ontology \mathcal{O} is said to be *satisfiable* or *consistent* if it has a model. An ontology that does not have a model is said to be *unsatisfiable* or *inconsistent* or *contradictory* [94, 103]. \diamond

Definition 3.16. A concept C is *subsumed* by a concept D with regards to an ontology \mathcal{O} if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model \mathcal{I} of \mathcal{O} . This is written as $C \sqsubseteq_{\mathcal{O}} D$ or $\mathcal{O} \models C \sqsubseteq D$ [8, 103]. *Classification* is the task of determining $\sqsubseteq_{\mathcal{O}}$ for all concept names in an ontology \mathcal{O} [6, 103]. \diamond

Definition 3.17. An assertion $C(a)$ is *satisfiable*, if and only if there exists an interpretation \mathcal{I} such that $a^{\mathcal{I}} \in C^{\mathcal{I}}$, where C is a concept and a is an individual [8]. An individual a is an *instance* of a concept C with respect to an ontology \mathcal{O} if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all interpretations \mathcal{I} of \mathcal{O} . This is written as $\mathcal{O} \models C(a)$ [94]. \diamond

These different reasoning tasks are all reducible to ontology consistency [8, 103]. Recall that an ontology is consistent if all the axioms in it are simultaneously satisfiable (Definition 3.15 on p. 46). Instance checking can be reduced to ontology consistency since $\mathcal{O} \models C(a)$ if and only if $\mathcal{O} \cup \{\neg C(a)\}$ is inconsistent [8, 36]. Subsumption (and by extension classification) can be reduced to concept satisfiability since $C \sqsubseteq D$ if and only if $C \sqcap \neg D$ is unsatisfiable [8]. Concept satisfiability can be reduced to axiom entailment since for an unsatisfiable concept $\mathcal{O} \models C \sqsubseteq \perp$ holds [103]. Furthermore, axiom entailment is reducible to ontology consistency by proof by contradiction, as follows: Let β be an axiom that claims the opposite of axiom α ². For the axioms α and β every interpretation either satisfies α or β , but not both. Now if $\mathcal{O} \models \alpha$ holds, no model of \mathcal{O} can be a model for β . Hence, the extended ontology $\mathcal{O}' = \mathcal{O} \cup \{\beta\}$ is inconsistent.

A benefit of the reducibility of these reasoning tasks is that it eases their implementation in tools. Rather than having to implement all the reasoning tasks, only for example ontology consistency can be implemented, and the rest of the reasoning tasks can be implemented based on their reduction to ontology consistency [103]. Tools that are available for doing consistency checking, concept satisfiability and classification are Hermit, FaCT++ and Pellet [35].

3.3.2 Other Reasoning Tasks

The reasoning tasks discussed in the previous section are deductive in nature. That is, logical consequences are inferred from explicitly stated information in an ontology. In contrast, the reasoning tasks, namely justification and abduction, discussed in this section are not deductive in nature.

Intuitively, abduction can be understood as the process through which a person will guess the reason(s) why certain facts are observed in the world. As an example, a person may note that it is wet outside and therefore conclude that it has rained. This conclusion may however be refuted when new information is learned, for instance, if the sprinklers were on. Abduction is formalized in Definition 3.18.

Definition 3.18. An *abductive problem* for an ontology \mathcal{O} and a logical consequence φ , represented by $\langle \mathcal{O}, \varphi \rangle$, is when φ is not entailed by \mathcal{O} , that is, $\mathcal{O} \not\models \varphi$. An axiom α is a *solution* to an abductive problem $\langle \mathcal{O}, \varphi \rangle$ if and only if $\mathcal{O} \cup \{\alpha\} \models \varphi$ [73]. \diamond

² β can be a single axiom or a set of axioms emulating the opposite of α . Corresponding opposite axioms are provided for *SRONTQ* in the provided reference (see p. 51 of [103]).

Research into abductive reasoning tasks in DLs is in its early stages. Currently a computational framework exists for abductive reasoning in the DL \mathcal{ALC} . However, this framework still needs to be extended to other DLs [74].

An inconsistent ontology is undesirable since it entails everything due to the principle of explosion [103]. Hence, no useful information can be inferred from it. Moreover, an inconsistent ontology is often indicative of modelling errors. This gave rise to the formulation of reasoning tasks that can explain (or justify) and correct modelling errors [98, 106].

A justification for an entailment is a minimal subset of an ontology such that the entailment still holds. A single justification for an entailment is calculated by removing axioms from the ontology until a minimal subset of the ontology remains such that the entailment still holds. A single entailment may have many justifications [56].

From a set of justifications $\{\mathcal{J}_1, \dots, \mathcal{J}_n\}$, a set \mathcal{R} can be constructed by adding a single axiom from each of the justifications $\{\mathcal{J}_1, \dots, \mathcal{J}_n\}$ to \mathcal{R} . Since the justifications are minimal, removing \mathcal{R} from \mathcal{O} is a repair for \mathcal{O} such that $\mathcal{O} \not\models \eta$ [56].

Justification and repair are defined formally in Definitions 3.19 and 3.20 respectively.

Definition 3.19. \mathcal{J} is a *justification* for $\mathcal{O} \models \eta$ if $\mathcal{J} \subseteq \mathcal{O}$, $\mathcal{J} \models \eta$, and for all $\mathcal{J}' \subsetneq \mathcal{J}$ it is the case that $\mathcal{J}' \not\models \eta$ [56]. \diamond

Definition 3.20. Given an ontology \mathcal{O} such that $\mathcal{O} \models \eta$, the set of axioms \mathcal{R} is a *repair* for η in \mathcal{O} if $\mathcal{R} \subseteq \mathcal{O}$, $\mathcal{O} \setminus \mathcal{R} \not\models \eta$ and there is no $\mathcal{R}' \subsetneq \mathcal{R}$ such that $\mathcal{O} \setminus \mathcal{R}' \not\models \eta$ [56]. \diamond

Pellet and RacerPro are reasoners (which can be used with both Protégé and Swoop [45]) that support the finding of justifications for inconsistent concepts [35], while the Swoop ontology editor has a plugin for doing repairs [102].

3.4 Summary

In this chapter an overview was given of the general syntax and semantics of DLs (see Section 3.1 on p. 31). In Section 3.2 (p. 40) the DLs \mathcal{ALCQI} and $\mathcal{SROIQ}^{(D)}$, as well as the OWL 2 Web Ontology Language, was introduced. In the next chapter existing research is discussed for translating UML class diagram features to \mathcal{ALCQI} and OWL 2, while in Chapter 5 (p. 60) $\mathcal{SROIQ}^{(D)}$ and OWL 2 are used to provide translations of UML class diagram features for the purpose of formal scenario testing.

Section 3.3 (p. 46) discussed reasoning tasks for DLs that are relevant to formal scenario testing. Ontology consistency checking is the core reasoning task that will be used in Chapters 6 (p. 92) and 7 (p. 119). Even though justifications are used to explain entailments in Chapters 6 (p. 92) and 7 (p. 119), current justification algorithms are not a perfect match for the needs of formal scenario testing. This, along with how abductive reasoning tasks can be of use in formal scenario testing, will be discussed in Section 6.2.4 (p. 102).

Chapter 4

DL Translations of UML Class Diagrams

In order to support the formal scenario testing of UML class diagrams, the diagrams can be translated to DLs. Substantial work has been done on the translation of UML class diagrams to DLs [5, 24, 25, 11, 10, 12, 13, 70, 4, 100]. UML class diagrams have been translated to the DLs \mathcal{ALC} , \mathcal{ALCI} , \mathcal{ALCQI} and \mathcal{DLR}_{ifd} [24, 13, 4, 100]. Artale, et. al. [5], showed that translating UML class diagrams and entity-relationship models to the DL-lite family of DL languages affords lower computational complexity bounds. More recently UML class diagrams have been translated to OWL 2 [118, 119]. As was explained in Section 1.3.4 (p. 7), this dissertation focuses on the DLs \mathcal{ALCQI} and $\mathcal{SROIQ}^{(D)}$. The DL \mathcal{ALCQI} is a subset of the DL $\mathcal{SROIQ}^{(D)}$ which serves as the mathematical logic basis of OWL 2 (see Section 3.2 on p. 40).

In this chapter the \mathcal{ALCQI} translation of UML class diagram notation is given as defined by Berardi, et. al. [13]. Where the related translation to OWL 2 was defined by Zedlitz, et. al. [118], the translation to OWL 2 is supplied as well. For UML class diagram features that cannot be expressed in \mathcal{ALCQI} , the $\mathcal{SROIQ}^{(D)}$ translation is given where it follows from existing research.

4.1 Classes

In this section the translation of classes with their attributes and operations to the DL \mathcal{ALCQI} and OWL 2 are given. A UML class C corresponds to an atomic concept C in \mathcal{ALCQI} and a class C in OWL 2.

An aspect in which UML class diagrams differ from DLs and OWL is that UML class diagrams make the UNA (see Section 3.1.6 on p. 39), while DLs and OWL 2 do not make this assumption. Thus, for DLs and OWL 2 uniqueness of names must be enforced for any pair of classes that are not in the same inheritance hierarchy. To make explicit that for example the classes C and T are disjoint, it is necessary to assert that the \mathcal{ALCQI} concepts C and T are disjoint. This is done through assertion (4.1).

$$C \sqsubseteq \neg T \tag{4.1}$$

To assert that the classes C and T are disjoint in OWL 2, it is sufficient to apply the `DisjointWith` axiom as shown in listing (4.2).

$$\begin{array}{l} \text{Class: } T \\ \text{DisjointWith: } C \end{array} \tag{4.2}$$

4.2 Attributes

Consider the class C with an attribute $\mathfrak{t}:T$ as illustrated in Figure 2.1(b) (p. 14). In *ALCQI* an attribute \mathfrak{t} is represented as an atomic role t . Assertion (4.3) is applied.

$$C \sqsubseteq \forall t.T \tag{4.3}$$

Berardi, et. al. [13], in order to simplify the translation, intentionally do not distinguish between classes and data types. Furthermore, *ALCQI* does not support concrete domains and nominals and hence, it is not possible to represent data types using *ALCQI* (see Section 3.2.1 (p. 40)) [7, 8, 13]. Data types will be discussed in more detail in Chapter 5 (p. 60).

Depending on whether the type T denotes a class or a data type, the attribute \mathfrak{t} is represented as an `ObjectProperty` or a `DataProperty` in OWL 2. Listing (4.4) defines the object property \mathfrak{t} with its domain and range.

The translation is similar where T is a data type, except that the `DataProperty` rather than the `ObjectProperty` axiom is used. In this dissertation assertions will be defined in terms of `ObjectProperty` axioms with the explicit understanding that it can be replaced with `DataProperty` axioms if data types rather than classes are used. Only where distinct differences between the use of object properties and data properties are present, will these differences be made explicit.

$$\begin{array}{l} \text{ObjectProperty: } \mathfrak{t} \\ \text{Domain: } C \\ \text{Range: } T \end{array} \tag{4.4}$$

4.2.1 Multiplicity

The multiplicity $[i..j]$ of an attribute t is expressed as

$$C \sqsubseteq (\geq i t. \top) \sqcap (\leq j t. \top) \quad (4.5)$$

in \mathcal{ALCQI} and as

$$\begin{aligned} \text{Class: } C \\ \text{SubClassOf: } (t \text{ min } i \text{ Thing}) \text{ and} \\ (t \text{ max } j \text{ Thing}) \end{aligned} \quad (4.6)$$

in OWL 2. When j is $*$, the second conjunct of the assertion is omitted. For $[0..*]$, the complete assertion is omitted. For the multiplicity $[1..1]$ the assertion

$$C \sqsubseteq \exists t. \top \sqcap (\leq 1 t. \top) \quad (4.7)$$

is applied in \mathcal{ALCQI} and for OWL 2 the assertion (4.8) is applied:

$$\begin{aligned} \text{Class: } C \\ \text{SubClassOf: } t \text{ exactly } 1 \text{ Thing} \end{aligned} \quad (4.8)$$

4.3 Operations

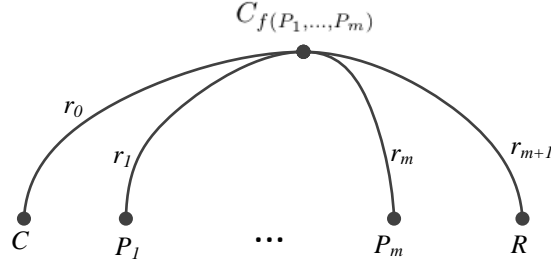
In this section the translation of UML class diagram operations to the DL \mathcal{ALCQI} is described. An equivalent translation of operations to OWL 2 is not supplied since, to the best knowledge of the author, operations have not been translated to OWL 2 as yet. However, as part of the contribution of this dissertation a translation of operations to OWL 2 is provided in Chapter 5 (p. 60).

Berardi, et. al. [13] distinguish between operations with no parameters and operations with m parameters. Operations with no parameters are addressed first and in Section 4.3.2 operations with parameters are addressed.

4.3.1 Operations with no Parameters

Consider the operation $f():R$ on class C , that is, the operation f taking no parameters and returning a value of type R . The operation f is modelled in \mathcal{ALCQI} as a role $r_{f()}$ for which the following assertion must hold:

$$C \sqsubseteq \forall r_{f()}. R \sqcap (\leq 1 r_{f()}. \top) \quad (4.9)$$

Figure 4.1: Reification of a $(m+2)$ -ary relation.

4.3.2 Operations with Parameters

Consider an operation $f(p_1:P_1, \dots, p_m:P_m):R$ taking one or more parameters. Hence, f is an operation taking m parameters p_1, \dots, p_m with types P_1, \dots, P_m respectively and returning a value of type R . Formally the operation $f(p_1:P_1, \dots, p_m:P_m):R$ corresponds with an $(m+2)$ -ary relation where the first component of the relation represents the class C , the next m components represent the parameters and the last component (that is, the component at $(m+2)$) represents the return type of the operation.

The $(m+2)$ -ary relation cannot be directly represented in \mathcal{ALCQI} , but instead needs to be represented using reification. The $(m+2)$ -ary relation can be reified by introducing the atomic concept $C_{f(P_1, \dots, P_m)}$ and $(m+2)$ roles $r_0, \dots, r_{(m+1)}$ such that every individual of $C_{f(P_1, \dots, P_m)}$ is linked to each component of the tuple $(x_0, \dots, x_{(m+1)})$ via the respective roles $r_0, \dots, r_{(m+1)}$ [26]. Reification is graphically represented in Figure 4.1. Hence, the operation $f(p_1:P_1, \dots, p_m:P_m):R$ is translated to \mathcal{ALCQI} by applying the following assertions using the reified atomic concept $C_{f(P_1, \dots, P_m)}$

$$\begin{aligned} C_{f(P_1, \dots, P_m)} &\sqsubseteq \exists r_0. \top \sqcap (\leq 1 r_0. \top) \sqcap \\ &\quad \vdots \\ &\quad \exists r_{(m+1)}. \top \sqcap (\leq 1 r_{(m+1)}. \top) \end{aligned} \quad (4.10)$$

$$C_{f(P_1, \dots, P_m)} \sqsubseteq \forall r_1. P_1 \sqcap \dots \sqcap \forall r_m. P_m \quad (4.11)$$

$$C \sqsubseteq \forall r_0^- . (C_{f(P_1, \dots, P_m)} \Rightarrow \forall r_{m+1}. R) \quad (4.12)$$

where $C_{f(P_1, \dots, P_m)} \Rightarrow \forall r_{m+1}. R$ is an abbreviation for $\neg C_{f(P_1, \dots, P_m)} \sqcup \forall r_{m+1}. R$ (see Table 3.7 on p. 41).

Assertion (4.10) ensures that each individual of $C_{f(P_1, \dots, P_m)}$ represents a tuple that is linked to each of the roles $r_0, \dots, r_{(m+1)}$. In DLs that do not have the tree-model property it is possible that there may be two or more different individuals of $C_{f(P_1, \dots, P_m)}$ representing the same tuple. \mathcal{ALCQI} has the tree-model property and hence it is not possible for different

individuals to represent the same tuple [13].

Assertion (4.11) enforces the correct typing of the parameters $p1:P1, \dots, pm:Pm$ based only on the name of the operation. The assertion (4.12) states that when operation f , represented by the atomic concept $C_{f(P_1, \dots, P_m)}$, is called on instances of class C it will return a value of type R .

4.4 Binary Associations

Consider the binary association of Figure 2.3 (p. 16). For \mathcal{ALCQI} the atomic role a is introduced (to represent the association A^1) along with assertion (4.13).

$$\top \sqsubseteq \forall a.T \sqcap \forall a^-.C \quad (4.13)$$

For OWL 2 the object properties a and a_inv , where a_inv is the inverse of a , are introduced. The relevant OWL 2 assertions are given in (4.14).

$$\begin{aligned} &\text{ObjectProperty: } a \\ &\quad \text{Domain: } C \\ &\quad \text{Range: } T \\ &\text{ObjectProperty: } a_inv \\ &\quad \text{Domain: } T \\ &\quad \text{Range: } C \\ &\quad \text{InverseOf: } a \end{aligned} \quad (4.14)$$

4.4.1 Multiplicity

The multiplicity $[i..j]$ of an association a is expressed as

$$\begin{aligned} C &\sqsubseteq (\geq i a.T) \sqcap (\leq j a.T) \\ T &\sqsubseteq (\geq k a^-.T) \sqcap (\leq l a^-.T) \end{aligned} \quad (4.15)$$

in \mathcal{ALCQI} and as

$$\begin{aligned} &\text{Class: } C \\ &\quad \text{SubClassOf: } (a \text{ min } i \text{ Thing}) \text{ and} \\ &\quad \quad (a \text{ max } j \text{ Thing}) \\ &\text{Class: } T \\ &\quad \text{SubClassOf: } (a_inv \text{ min } k \text{ Thing}) \text{ and} \\ &\quad \quad (a_inv \text{ max } l \text{ Thing}) \end{aligned} \quad (4.16)$$

in OWL 2.

¹ a is preferred to A which agrees with the naming convention stated in Section 3.1.1 (p. 31). a in this context represents a role and not an individual.

4.5 Generalization/Specialization of Classes

Figure 2.6 (p. 18) refers. Stating that UML class C_1 is a subclass of class C is expressed as

$$C_1 \sqsubseteq C \quad (4.17)$$

in *ALCQI* and in OWL 2 as

$$\begin{array}{l} \text{Class: } C_1 \\ \text{SubClassOf: } C \end{array} \quad (4.18)$$

To state that the subclasses C_1, \dots, C_n of class C cover class C , assertions (4.19) are added for *ALCQI*.

$$C \sqsubseteq C_1 \sqcup \dots \sqcup C_n \quad (4.19)$$

For OWL 2 assertion (4.20) is added.

$$\begin{array}{l} \text{Class: } C \\ \text{SubClassOf: } C_1 \text{ or } \dots \text{ or } C_n \end{array} \quad (4.20)$$

To state that classes C_1, \dots, C_n are disjoint the assertions of (4.21) suffice for *ALCQI*.

$$C_i \sqsubseteq \prod_{j=i+1}^n \neg C_j \quad \text{for } 1 \leq i \leq n-1 \quad (4.21)$$

In OWL 2 this is expressed as

$$\text{DisjointClasses: } C_1, \dots, C_n \quad (4.22)$$

When the subclasses C_1, \dots, C_n are disjoint with each other and they cover class C (that is, the UML annotation `{complete, disjoint}` is applied) it can be abbreviated in OWL 2 as

$$\begin{array}{l} \text{Class: } C \\ \text{DisjointUnionOf: } C_1, \dots, C_n \end{array} \quad (4.23)$$

4.6 Association Specialization

Assuming the object properties `a1`, `a1_inv`, `a2` and `a2_inv` have been defined as for binary associations (see Section 4.4 (p. 53)), the assertions of (4.24) need to be added to transcribe the association specialization of Figure 2.7(a) (p. 19).

$$\begin{array}{l}
\text{ObjectProperty: } a_2 \\
\text{SubPropertyOf: } a_1 \\
\text{ObjectProperty: } a_{2_inv} \\
\text{SubPropertyOf: } a_{1_inv}
\end{array}
\tag{4.24}$$

Association specialization cannot be translated to \mathcal{ALCQI} since it does not have a role inclusion constructor (see Section 3.2.1 (p. 40)). However, as discussed in Section 3.2.2 (p. 41), $\mathcal{SROIQ}^{(D)}$ does have a role inclusion constructor. The translation of association specialization for DLs with role inclusion constructors are given by Calvanese, et. al. [27].

$$a_2 \sqsubseteq a_1 \tag{4.25}$$

4.7 Data Types

In this dissertation only primitive data types and enumerations are used. User-defined data types are not explicitly considered due to scope constraints. Since \mathcal{ALCQI} does not have constructors for concrete domains and nominals [7, 8, 13], it is not possible to represent data types in \mathcal{ALCQI} .

Zedlitz, et. al. [119] discuss in detail the translation of UML data types to OWL 2. Predefined primitive data types are translated to the corresponding XML schema data types that underpins OWL 2. The `Colour` enumeration of Figure 2.2 (p. 15) cannot be translated to OWL 2 due to the presence of the operation. If, for the moment, the operation is ignored, the `Colour` enumeration is translated as follows:

$$\begin{array}{l}
\text{DataType: } \text{Colour} \\
\text{EquivalentTo: "Red", "Amber", "Green"}
\end{array}
\tag{4.26}$$

In Section 5.4.1 (p. 73) the $\mathcal{SROIQ}^{(D)}$ and OWL 2 translations of enumerations with operations will be given.

4.8 Summary

To conclude this chapter, a summary is provided of the translations of UML class diagram features to \mathcal{ALCQI} (or where applicable $\mathcal{SROIQ}^{(D)}$) and OWL 2 in Table 4.1 (p. 59).

Table 4.1: UML class diagram translation to *ALCQI* and OWL 2


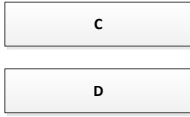
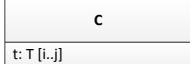
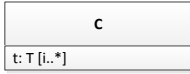

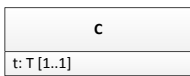

UML class diagram feature	<i>ALCQI</i>	OWL 2	Ref.
 <p>Class C</p>	C	Class: C	4.1 p. 49
 <p>Classes C and D</p>	$C \sqsubseteq \neg D$	Class: C Class: D DisjointWith: C	4.1 p. 49
 <p>Attribute t of type T with multiplicity [i..j]</p>	$C \sqsubseteq \forall t.T$ $C \sqsubseteq (\geq i t.T) \sqcap (\leq j t.T)$	Class: C SubClassOf: (t min i Thing) and (t max j Thing) Class: T ObjectProperty: t Domain: C Range: T	4.2 p. 50
 <p>Attribute t of type T with multiplicity [i..*]</p>	$C \sqsubseteq \forall t.T$ $C \sqsubseteq (\geq i t.T)$	Class: C SubClassOf: (t min i Thing) Class: T ObjectProperty: t Domain: C Range: T	4.2.1 p. 51
 <p>Attribute t of type T with multiplicity [0..*]</p>	$C \sqsubseteq \forall t.T$	Class: C Class: T ObjectProperty: t Domain: C Range: T	4.2.1 p. 51
 <p>OR</p>  <p>Attribute t of type T with multiplicity [1..1]</p>	$C \sqsubseteq \forall t.T$ $C \sqsubseteq \exists t.T \sqcap (\leq 1 t.T)$	Class: C SubClassOf: t exactly 1 Thing Class: T ObjectProperty: t Domain: C Range: T	4.2.1 p. 51

Table 4.1: UML class diagram translation to \mathcal{ALCQI} and OWL 2


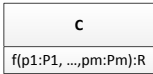
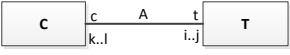
UML class diagram feature	\mathcal{ALCQI}	OWL 2	Ref.
 <p>Operation f with no parameters returning a value of type R</p>	$C \sqsubseteq \forall r_{f()}.R \sqcap (\leq 1r_{f()}.T)$	Not translated as yet	4.3 p. 51
 <p>Operation f with parameters p_1, \dots, p_m respectively of type P_1, \dots, P_m returning a value of type R</p>	$C_{f(P_1, \dots, P_m)} \sqsubseteq$ $\exists r_0.T \sqcap (\leq 1r_0.T) \sqcap$ \vdots $\exists r_{(m+1)}.T \sqcap (\leq 1r_{(m+1)}.T)$ $C_{f(P_1, \dots, P_m)} \sqsubseteq$ $\forall r_1.P_1 \sqcap \dots \sqcap \forall r_m.P_m$ $C \sqsubseteq$ $\forall r_0^-. (C_{f(P_1, \dots, P_m)} \Rightarrow \forall r_{m+1}.R)$	Not translated as yet	4.3 p. 51
 <p>Association A exists between classes C and T</p>	$T \sqsubseteq \forall a.T \sqcap \forall a^-.C$ $C \sqsubseteq (\geq i a.T) \sqcap (\leq j a.T)$ $T \sqsubseteq (\geq k a^-.T) \sqcap (\leq l a^-.T)$	ObjectProperty: a Domain: C Range: T ObjectProperty: a_inv Domain: T Range: C InverseOf: a Class: C SubClassOf: (a min i Thing) and (a max j Thing) Class: T SubClassOf: (a_inv min k Thing) and (a_inv max l Thing)	4.4 p. 53

Table 4.1: UML class diagram translation to *ALCQI* and OWL 2

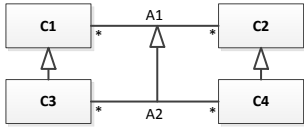
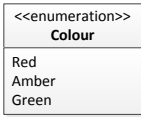
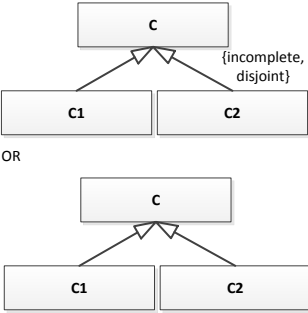
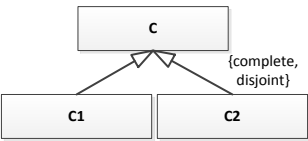
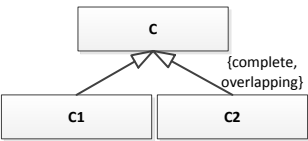
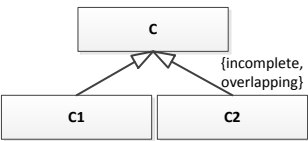
UML class diagram feature	<i>ALCQI</i>	OWL 2	Ref.
 <p>Association A2 specializes association A1</p>	<p>The <i>SROIQ(D)</i> translation is</p> $\top \sqsubseteq \forall a_1. C_2 \sqcap \forall a_1^-. C_1$ $\top \sqsubseteq \forall a_2. C_4 \sqcap \forall a_2^-. C_3$ $a_2 \sqsubseteq a_1$	<pre> Class: C1 Class: C2 Class: C3 Class: C4 ObjectProperty: a1 Domain: C1 Range: C2 ObjectProperty: a1_inv Domain: C2 Range: C1 ObjectProperty: a2 Domain: C3 Range: C4 SubPropertyOf: a1 ObjectProperty: a2_inv Domain: C4 Range: C3 SubPropertyOf: a1_inv </pre>	4.6 p. 54
 <p>The Colour enumeration consists of the colours Red, Amber and Green</p>	Not translated as yet	<pre> DataType: Colour EquivalentTo: "Red", "Amber", "Green" </pre>	4.7 p. 55
 <p>Class C is specialized by the disjoint classes C1 and C2 which do not cover class C</p>	$C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$ $C_1 \sqsubseteq \neg C_2$	<pre> Class: C Class: C1 SubClassOf: C Class: C2 SubClassOf: C DisjointClasses: C1, C2 </pre>	4.5 p. 54

Table 4.1: UML class diagram translation to *ALCQI* and OWL 2

UML class diagram feature	<i>ALCQI</i>	OWL 2	Ref.
 <p>Class C is specialized by the disjoint classes C1 and C2 which cover class C</p>	$C \sqsubseteq C_1 \sqcup C_2$ $C_1 \sqsubseteq \neg C_2$ $C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	<pre>Class: C DisjointUnionOf: C1, C2 Class: C1 SubClassOf: C Class: C2 SubClassOf: C</pre>	4.5 p. 54
 <p>Class C is specialized by the overlapping classes C1 and C2 which cover class C</p>	$C \sqsubseteq C_1 \sqcup C_2$ $C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	<pre>Class: C SubClassOf: C1 or C2 Class: C1 SubClassOf: C Class: C2 SubClassOf: C</pre>	4.5 p. 54
 <p>Class C is specialized by the overlapping classes C1 and C2 which do not cover class C</p>	$C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	<pre>Class: C Class: C1 SubClassOf: C Class: C2 SubClassOf: C</pre>	4.5 p. 54

Chapter 5

OWL and DL Translations for Scenario Testing

In order to perform formal scenario testing, the existing DL/OWL translations of UML class diagrams are extended. The previous chapter discussed features of UML class diagrams that have already been translated to either \mathcal{ALCQI} or OWL 2. This chapter extends the existing translations in the following ways:

1. Features of UML class diagrams that are required for formal scenario testing that have not previously been translated to any DL related formalisms, are translated. Identity constraints on UML class diagrams have not been translated to DL or OWL as yet (Section 5.1). Some features of operations have not been translated to DLs or OWL 2 and these are addressed in Section 5.3 (p. 66)
2. Berardi, et. al [13] made the assumption that the same attribute/association and operation can be shared across classes. This resulted in the domains and ranges of the relevant roles of the translations of attributes/associations and operations to \mathcal{ALCQI} to be overly lenient. In Section 5.2 (p. 63) the domains and ranges of these roles are defined tightly.
3. Operations have not been translated to OWL 2 before. A translation for operations to OWL 2 is provided in Section 5.3 (p. 66).
4. The translation of enumerations and binary associations are revisited to optimize modeller productivity in Section 5.4 (p. 73).
5. In Section 2.1.5 (p. 18) it was stated that subsetting and redefinition of association ends are forms of association specialization. In Chapter 4 (p. 49) the $\mathcal{SROIQ}^{(D)}$ and OWL 2 translations of association specialization were given (see Section 4.6 (p. 54)). However, since association specialization is defined for associations while subsetting and redefinition are defined for association ends, the translations for subsetting and redefinition of association ends were not made explicit. In Section 5.5 (p. 75) the translations of these are made explicit.

6. In order to be able to detect the various heuristics mentioned in Section 2.2 (p. 21) using formal scenario testing, the translation of UML class diagrams is refined with this specific purpose in mind. The refinement of the translation of operations and how formal scenario testing is dealing with the unique names of UML class diagram features are handled in Section 5.3 (p. 66) and Section 5.7 (p. 76).
7. The equivalence of attributes and associations are confirmed in Section 5.6 (p. 75).
8. Section 5.8 (p. 79) provides the rationale for excluding aggregation and composition from formal scenario testing.

This chapter concludes with a summary of the contributions of this chapter and a review of related research (Section 5.9 (p. 80)).

5.1 UML Class Diagram Identity Constraints

Identity constraints on UML class diagrams have been added in version 2.4.1 of the UML specification [64]. Identity constraints on UML class diagrams have not been translated to DLs or OWL 2 as yet. It is the aim of this section to provide a translation of identity constraints on UML class diagrams that can be used for the purpose of formal scenario testing.

Section 5.1.2 (p. 62) explains some of the challenges in translating identity constraints on UML class diagrams to DLs, as well as the Easy Key identity constraint implementation of OWL 2. In designing Easy Keys, a number of compromises were made. In Section 5.1.3 (p. 62) these compromises are evaluated in the context of formal scenario testing. However, before any attempt can be made to provide a translation of identity constraints on UML class diagrams, the exact meaning of identity constraints has to be clarified. This is the topic of Section 5.1.1.

5.1.1 Problematic Interpretation of Compound Keys

With regards to compound keys the UML specification states the following [64]:

“If multiple properties are marked (possibly in superclasses), then it is the combination of the (property, value) tuples that will logically provide the uniqueness for any instance.”

Consider the example given in Figure 2.8(c) on p. 20. The `{id}` property modifier on the attribute `idNumber` of the `SACitizen` class implies that instances of the `SACitizen` class are uniquely identified by their `idNumber`. According to the quote from the UML specification above, the `{id}` property modifier on the `employeeCode` attribute cannot be considered on its own, but needs to be considered in conjunction with the `{id}` property modifiers

on the superclass. Hence, according to the UML specification, instances of the `Employee` class are uniquely identified exclusively by the compound key consisting of `idNumber` and `employeeCode`. This leads to a contradiction since instances of `Employee` are also instances of `SACitizen`.

A more appropriate interpretation is that instances of the `Employee` class can be uniquely identified by either `idNumber` or `employeeCode` or the compound key consisting of `idNumber` and `employeeCode`. For the purpose of this dissertation, this is the interpretation that will be used for `{id}` property modifiers applied across an inheritance hierarchy.

5.1.2 Identity Constraint Challenges and OWL 2 Easy Keys

The challenge of mapping UML identity constraints to DLs is that identity constraints in UML class diagrams are usually applied to data types. It is a well-known fact that identity constraints in the presence of data types leads to undecidability even for the DL \mathcal{ALC} extended accordingly [84, 97].

The semantics of OWL 2 is based on the DL $\mathcal{SROIQ}^{(D)}$. For the implementation of Easy Keys, a relaxation of DL-safe rules was incorporated into OWL 2. Decidability is maintained by restricting reasoning on identity constraints to named individuals. With the Easy Key implementation both concepts and data types can serve as keys of concepts [97].

The Easy Key translations for examples (a) - (c) of Figure 2.8 (p. 20) are given in listings (5.1) - (5.3).

```
Class: SACitizen
    HasKey: idNumber
```

(5.1)

```
Class: Employee
    HasKey: idNumber, employeeCode
```

(5.2)

```
Class: SACitizen
    HasKey: idNumber
Class: Employee
    SubClassOf: SACitizen
    HasKey: employeeCode
    HasKey: idNumber, employeeCode
```

(5.3)

5.1.3 The Effect of Easy Keys Compromises on Formal Scenario Testing

Parsia, et. al. state that keys in general have (or may have) the following properties [97]:

(P1) If two individuals x and y have the same key values, then $x = y$.

(P2) Integrity constraint: missing key values raise an error. Individuals which may have a key must have a key. This is easily seen in the case of relational database rows.

(P3) Functionality constraint: entities have only one key. This can be interpreted weakly (in any model, a given entity has at most one key) or strictly (each entity has a known key, the same one in every model).

For the implementation of P1, Easy Keys restrict reasoning to individuals explicitly specified in the ontology (as opposed to generated individuals) in an attempt to keep reasoning on identity constraints feasible. Formal scenario testing is concerned with named individuals (as is explained in Section 6.1 on p. 92) and is therefore not affected by this restriction in Easy Keys.

P2 is not expressible in first-order logic or OWL 2, due to P2 being non-monotonic, consequently its implementation in Easy Keys has been forgone [97]. Forgoing the implementation of property P2 is compatible with the requirements of formal scenario testing. The purpose of identity constraints in formal scenario testing is to enable the detection of various modelling heuristic violations (see Chapter 7 (p. 119)). It is not the intent to use identity constraints in formal scenario testing as a means of enforcing integrity constraints on data. Rather, data for industry-strength software is typically stored in a relational database, which by definition, has support for integrity constraints. Furthermore, for many formal scenario tests identity constraints will not be the focus of the test. Due to the absence of P2 the modeller is not forced to supply values for identity constraints, which results in the more efficient use of the modeller's time. However, when the modeller wants to specifically test identity constraints, i.e. that two individuals do not have the same key, the Easy Keys implementation will highlight consistencies/inconsistencies.

Only the weak interpretation of property P3 is implemented in OWL 2, which states that in any model a given entity has at most one key value. Formally, a particular business domain is just one possible interpretation of an ontology. If this specific interpretation satisfies the ontology, it is a model for the ontology. Business is only concerned with their particular interpretation of an ontology and therefore the strict interpretation of P3 is not required by businesses. Hence the weak interpretation of P3 is sufficient for formal scenario testing.

5.2 Tight Specification of Domain and Range Restrictions

In the translations of UML class diagram features to *ALCQI* and OWL 2 of Chapter 4 (p. 49) there are a disconnect between how UML features are translated to *ALCQI* versus OWL 2:

- In translating attributes/associations and operations to *ALCQI* various roles are introduced. The domains and ranges of these roles are not constrained sufficiently to

accurately represent the UML semantics of these features.

- For binary associations translated to *ALCQI* the domain and range restrictions are given as a single assertion (see (4.13) on p. 53). In OWL 2 domain and range restrictions are stated separately. In an attempt to make the translation from *ALCQI/SROIQ^(D)* to OWL 2 more apparent, the preference in this dissertation will be to split domain and range restrictions for *ALCQI/SROIQ^(D)* translations as well.

In this section the DL translations of UML class diagrams will be provided based on the domain and range restrictions as defined in a presentation by De Giacomo [44], which is based on research by Calvanese, et. al. [27]. This section is structured as follows:

1. Section 5.2.1 explains why the attribute translation of Berardi, et. al. [13], is incorrect and it provides the correct translation to *SROIQ^(D)*. This correction also applies to binary associations, which are addressed in Section Section 5.2.2 (p. 65).
2. Berardi, et. al. [13], made a similar error in the translation for operations as for attributes. Furthermore, Berardi, et. al. [13], provided dissimilar translations for operations with parameters and operations without parameters. These aspects are discussed in Section 5.2.3 (p. 65).

5.2.1 Attributes

Berardi, et. al. [13], assumed that an attribute *a* appearing in two different classes, *C1* and *C2*, of possibly different types, are in actual fact the same attribute. This assumption is incorrect due the use of qualified names in UML as explained in Section 2.1.7 (p. 21). This incorrect assumption of Berardi, et. al. [13], resulted in the translation (4.3) on p. 50, which in essence states that *C* is a subset of the domain of role *a* and *T* is a subset of the range of role *a*¹. This assertion falls short of the UML semantics for attributes, which state that the domain of attribute *a* is class *C* and the range is *T* [64].

The intended semantics of attributes are correctly captured by the translation of Zedlitz, et. al. in assertion (4.4) on p. 50 [118]. This assertion states that the domain and range of a property *t* is respectively the class *C* and the class (or data type) *T*. This translation also captures correctly the semantics of UML class diagrams where a class *C1* extends class *C* on which attribute *a* of type *T* is defined. Since in OWL 2 class *C1* is defined as a subclass (or subset in terms of DLs) of class *C*, it follows that the domain and range of the inherited attribute *a* in class *C1* is the same as that of the attribute *a* of the parent class *C*.

Based on the preceding discussion, this dissertation gives preference to the translation of attributes as given by Zedlitz, et. al [118]. This is also the translation supplied by De

¹Note that in terms of the assumption made by Berardi, et. al. [13], axiom (4.3) on p. 50 is correct. However, the contention here is that the Berardi, et. al. [13] assumption is incorrect.

Giacomo [44]. Hence, an attribute a of type T of class C is translated to $\mathcal{SROIQ}^{(D)}$ as

$$\exists t.\top \sqsubseteq C \tag{5.4}$$

$$\exists t^{\neg}.\top \sqsubseteq T \tag{5.5}$$

where for role t assertion (5.4) represents the domain and assertion (5.5) represents the range.

5.2.2 Binary Associations

According to the UML specification, a binary association expresses a relation between two specific types. These constraints on the types of a binary association is not captured correctly by (4.13) on p. 53. Similar to Calvanese, et. al. [27], the preference in this dissertation is to use assertions

$$\exists a.\top \sqsubseteq C \tag{5.6}$$

$$\exists a^{\neg}.\top \sqsubseteq T \tag{5.7}$$

where (5.6) represents the domain and (5.7) represents the range of role a for association A .

5.2.3 Operations

As mentioned in the introduction of this section (Section 5.2), there are two aspects with regards to operations that are addressed here:

1. Berardi, et. al. [13], in their translation of operations do not take cognizance of qualified names in UML.
2. Berardi, et. al. [13], provided dissimilar translations for operations with parameters and operations without parameters.

Berardi, et. al. [13], assumed that an operation $f(p_1:P_1, \dots, p_m:P_m):R$ defined for classes C_1 and C_2 represent the same operation with the same parameters. As discussed in Section 2.1.7 (p. 21), due to the use of qualified names in UML, these two operations in UML have different names as well as different parameter names. Taking the qualified names of UML into consideration, the domains and ranges of the roles of assertions (4.11) and (4.12) on p. 52 are overly lenient. To ensure that the roles introduced in the translation of operations represent the semantics of UML accurately, assertions (5.8) and (5.9) are preferred to assertions (4.11) and (4.12) on p. 52. Hence, $C_{f(P_1, \dots, P_m)}$ is the domain and P_i are the ranges for the roles r_i where $i = 1, \dots, m$ in assertion (5.8). In assertion (5.9) $C_{f(P_1, \dots, P_m)}$ is the domain, and C and R are the ranges for roles r_0 and r_{m+1} respectively.

$$\exists r_i.\top \sqsubseteq C_{f(P_1, \dots, P_m)} \quad \exists r_i^-. \top \sqsubseteq P_i \quad i = 1, \dots, m \quad (5.8)$$

$$\begin{aligned} \exists r_0.\top &\sqsubseteq C_{f(P_1, \dots, P_m)} & \exists r_0^-. \top &\sqsubseteq C \\ \exists r_{m+1}.\top &\sqsubseteq C_{f(P_1, \dots, P_m)} & \exists r_{m+1}^-. \top &\sqsubseteq R \end{aligned} \quad (5.9)$$

For assertion (4.9) on p. 51 the domain and range of role $r_{f()}$ is also overly lenient. In assertions (5.10) and (5.11) the domain and range of role $r_{f()}$ is specified tightly enough to reflect the semantics of the UML specification precisely [64]. Assertion (5.12) ensures that evoking operation $f()$ on class C is deterministic.

$$\exists r_{f()}. \top \sqsubseteq C \quad (5.10)$$

$$\exists r_{f()}^-. \top \sqsubseteq R \quad (5.11)$$

$$C \sqsubseteq (\leq 1 r_{f()}. \top) \quad (5.12)$$

Note that it is indeed possible to express assertions (5.10) – (5.12) as a single assertion, but in this dissertation the preference is to specify the various aspects of the semantics of UML as separate assertions. Using separate assertions make it easier to distinguish the effect of the nuances of various UML features.

5.3 Operations

This section addresses a number of aspects regarding the translation of operations.

1. The translation of Berardi, et al. [13] uses arbitrary roles names in translating operations. Section 5.3.1 refines the translation of operations to make use of an explicit naming convention.
2. Section 5.3.2 (p. 68) refines the translation of operations to enforce the constraint that a class that defines an operation must be able to call it.
3. Berardi, et al. [13] translate operations with no parameters and operations taking one or more parameters differently. Section 5.3.3 (p. 69) provides a translation of operations with no parameters, which is similar to the translation of operations with one or more parameters.

4. Since the translation of operations to OWL 2 has not been done before, the related OWL 2 translation is provided in Section 5.3.4 (p. 69).
5. When translating operations with return values it is necessary to ensure that the combination of class instance and parameter values will determine the return value uniquely. In *ACLQI* no specific assertions are required to enforce this constraint since *ACLQI* has the tree-model property [13]. However, *SROIQ^(D)* and OWL 2 do not have the tree-model property [61, 76] and therefore specific assertion(s) need to be provided to enforce unique return values. This is considered in Section 5.3.5 (p. 71).
6. A translation for operations that do not return a value is given in Section 5.3.6 (p. 71).

5.3.1 Explicit Naming Convention

The existing translation of the operation $f(p_1:P_1, \dots, p_m:P_m):R$ by Berardi, et al. [13] introduces the atomic concept $C_{f(P_1, \dots, P_m)}$ and arbitrary roles r_0, \dots, r_{m+1} for the reified representation of the $(m+2)$ -ary relation as was explained in Section 4.3 (p. 51).

Firstly, to make explicit that the operation $f(p_1:P_1, \dots, p_m:P_m):R$ returns a value of type R , the atomic concept $C_{f(P_1, \dots, P_m):R}$ will be used rather than $C_{f(P_1, \dots, P_m)}$. Adding the return type to the concept name, makes it possible to distinguish between concepts representing operations that have a return value and operations that do not have a return value. Operations that have no return values will be discussed in Section 5.3.6 (p. 71).

Secondly, rather than introducing arbitrary roles, the use of roles that make their intended application explicit, is proposed. The role f^- will be used rather than r_0 to associate individuals of C with individuals of the atomic concept $C_{f(P_1, \dots, P_m):R}$. The reason for using the inverse of f is discussed in the next section. Furthermore, the roles p_1, \dots, p_m are preferred (as opposed to the roles r_1, \dots, r_m) where the role names correspond with the names of the parameters of the operation f . Thus, the roles p_1, \dots, p_m are used to link individuals of the concept $C_{f(P_1, \dots, P_m):R}$ with individuals of the concepts P_1, \dots, P_m respectively.

Lastly, for the role that assigns the return value of type R to the atomic concept $C_{f(P_1, \dots, P_m):R}$, the role r_R is used to make the return type explicit. Roles with explicit names rather than arbitrary roles will be used in Chapter 7 (p. 119) to identify operations that share the same signature or partial signature. The assertions (4.10), (5.8) and (5.9) are rewritten as follows:

$$\begin{aligned}
C_{f(P_1, \dots, P_m):R} &\sqsubseteq \exists f^-. \top \sqcap (\leq 1f^-. \top) \sqcap \\
&\quad \exists p_1. \top \sqcap (\leq 1p_1. \top) \sqcap \\
&\quad \vdots \\
&\quad \exists p_m. \top \sqcap (\leq 1p_m. \top) \sqcap \\
&\quad \exists r_R. \top \sqcap (\leq 1r_R. \top)
\end{aligned} \tag{5.13}$$

$$\exists p_i. \top \sqsubseteq C_{f(P_1, \dots, P_m):R} \quad \exists p_i^-. \top \sqsubseteq P_i \quad i = 1, \dots, m \tag{5.14}$$

$$\begin{aligned}
\exists f^-. \top &\sqsubseteq C_{f(P_1, \dots, P_m):R} & \exists f. \top &\sqsubseteq C \\
\exists r_R. \top &\sqsubseteq C_{f(P_1, \dots, P_m):R} & \exists r_R^-. \top &\sqsubseteq R
\end{aligned} \tag{5.15}$$

5.3.2 An Operation is Performed by the Class that Defines it

Defining an operation on a class in essence states that instances of the class must in general be able to perform the behaviour as defined by the operation² [18, 64, 104]. The translation of Berardi, et. al. [13] do not make this constraint explicit. This constraint is formalized here.

For a class C with an operation $f(p_1:P_1, \dots, p_m:P_m):R$ the atomic concept $C_{f(P_1, \dots, P_m):R}$ and the role f^- (along with the roles p_1, \dots, p_m, r_R) are introduced as was explained in the previous section. Assertion (5.16) is added to the assertions (5.13)-(5.15).

$$C \sqsubseteq \exists f. C_{f(P_1, \dots, P_m):R} \tag{5.16}$$

Assertion (5.16) states that every individual of C is linked via the role f to at least one individual of $C_{f(P_1, \dots, P_m):R}$. Since operation f can be performed on instances of class C the preference is to use the role f such that the concept C is the domain of f . This is the reason for using the role f^- in the reification of the $(m+2)$ -ary relation using atomic concept $C_{f(P_1, \dots, P_m):R}$ (as was mentioned in Section 5.3.1 on p. 67). The equivalent assertion of (5.16) is expressed in OWL 2 as assertion (5.17).

²It is possible to specify preconditions on an operation which give an indication as to when it is permitted to call an operation on an instance of a class. However, in this dissertation preconditions are not considered due to scope constraints. Therefore the assumption is that when an operation is defined on a class, it must be possible to call the operation.

$$\begin{aligned} \text{Class: } C & & (5.17) \\ \text{SubClassOf: } f \text{ some } C_f(P1, \dots, Pm)_R & \end{aligned}$$

5.3.3 Operations with No Parameters

In line with the translation of operations with one or more parameters, a translation for an operation taking zero parameters (see assertion (4.9) on p. 51 of Section 4.3) is provided based on reification. The atomic concept $C_{f():R}$ and the roles f^- and r_R are introduced and assertions (5.18) and (5.19) are applied:

$$\begin{aligned} C_{f():R} \sqsubseteq \exists f^-. \top \sqcap (\leq 1 f^-. \top) \sqcap & \\ \exists r_R. \top \sqcap (\leq 1 r_R. \top) & \end{aligned} \quad (5.18)$$

$$\begin{aligned} \exists f^-. \top \sqsubseteq C_{f():R} & \quad \exists f. \top \sqsubseteq C & (5.19) \\ \exists r_R. \top \sqsubseteq C_{f():R} & \quad \exists r_R^-. \top \sqsubseteq R \end{aligned}$$

Assertion (5.18) ensures that each individual of the atomic concept $C_{f():R}$ is linked to the roles f^- and r_R while assertions (5.19) enforces the return type of operation $f()$ on class C as being of type R .

5.3.4 OWL 2 Translation of Operations

In translating operations for the DLs *ALCQI* and *SRQIQ^(D)* the atomic concept $C_{f(P_1, \dots, P_m):R}$ and the roles $f^-, p_1, \dots, p_m, r_R$ have been introduced as was explained in Section 5.3.1 (p. 67). Similarly for OWL 2 the class $C_f(P1, \dots, Pm)_R$ and the properties $f_inv, p1, \dots, pm, r_R$ are introduced. The only explicit inverse property that is required is the inverse of f which is defined as f_inv . Both the properties f and f_inv will be essential in applying formal scenario testing, which is discussed in Chapter 7 (p. 119).

Thus the assertions (5.13) - (5.15) in OWL 2 becomes assertions (5.20) - (5.22):

```

ObjectProperty: f_inv
ObjectProperty: p1
      ⋮
ObjectProperty: pm
ObjectProperty: r_R
Class: C_f(P1, ..., Pm)_R
      SubClassOf:
          f_inv exactly 1 Thing and
          p1 exactly 1 Thing and
          ⋮
          pm exactly 1 Thing and
          r_R exactly 1 Thing

```

(5.20)

```

ObjectProperty: p1
      Domain: C_f(P1, ..., Pm)_R
      Range: P1
      ⋮
ObjectProperty: pm
      Domain: C_f(P1, ..., Pm)_R
      Range: Pm

```

(5.21)

```

ObjectProperty: f_inv
      Domain: C_f(P1, ..., Pm)_R
      Range: C
ObjectProperty: f
      InverseOf: f_inv
ObjectProperty: r_R
      Domain: C_f(P1, ..., Pm)_R
      Range: R

```

(5.22)

For the operation $f():R$ the class $C_f()_R$ and the properties f_inv , f and r_R are introduced and the assertions (5.23) and (5.24) are applied:

```

ObjectProperty: f_inv
ObjectProperty: r_R
Class: C_f()_R
      SubClassOf:
          f_inv exactly 1 Thing and
          r_R exactly 1 Thing

```

(5.23)

$$\begin{array}{l}
\text{ObjectProperty: } f_inv \\
\text{Domain: } C_f()_R \\
\text{Range: } C \\
\text{ObjectProperty: } f \\
\text{InverseOf: } f_inv \\
\text{ObjectProperty: } r_R \\
\text{Domain: } C_f()_R \\
\text{Range: } R
\end{array} \tag{5.24}$$

5.3.5 Unique Return Values

In order to ensure determinism of an operation $f(p1:P1, \dots, pm:Pm):R$ it is necessary to ensure that the return value of type R will be uniquely determined for a given instance of class C with given parameter values $p1, \dots, pm$. In *ALCQI* this is achieved without additional assertions due to the fact that *ALCQI* has the tree-model property [13]. *SROIQ^(D)* and OWL 2 do not have the tree-model property [61, 76]. Thus, other means need to be considered for achieving determinism of operations.

For the DL *DL \mathcal{R}_{ifd}* the determinism of operations with return values is achieved through applying an identity constraint [13]. Section 5.1 (p. 61) discussed how identity constraints can be applied in OWL 2 using Easy Keys [97]. Easy Keys are not directly expressible in *SROIQ^(D)* but instead rely on the extension of *SROIQ^(D)* with DL safe rules. Hence, only the OWL 2 translation is provided here.

Assertion (5.25) applies the identity constraint to the properties $f_inv, p1, \dots, pm$ on the class $C_f(P1, \dots, Pm)_R$. It states that two or more different individuals of class $C_f(P1, \dots, Pm)_R$ cannot agree on their participation in properties $f_inv, p1, \dots, pm$.

$$\begin{array}{l}
\text{Class: } C_f(P1, \dots, Pm)_R \\
\text{HasKey: } f_inv, p1, \dots, pm
\end{array} \tag{5.25}$$

For the operation $f():R$ with class $C_f()_R$ and property f_inv assertion (5.26) is added.

$$\begin{array}{l}
\text{Class: } C_f()_R \\
\text{HasKey: } f_inv
\end{array} \tag{5.26}$$

5.3.6 Operations with no Return Values

The UML specification allows operations of the form $f(p1:P1, \dots, pm:Pm)$, which defines an operation that has no return value [64]. Here the translation of such operations are made

explicit. This is achieved by removing reference to the role r_R from (5.13) and (5.15) as shown in (5.27) and (5.28) respectively:

$$\begin{aligned}
C_{f(P_1, \dots, P_m)} \sqsubseteq & \exists f^-. \top \sqcap (\leq 1 f. \top) \sqcap \\
& \exists p_1. \top \sqcap (\leq 1 p_1. \top) \sqcap \\
& \vdots \\
& \exists p_m. \top \sqcap (\leq 1 p_m. \top)
\end{aligned} \tag{5.27}$$

$$\exists f^-. \top \sqsubseteq C_{f(P_1, \dots, P_m)} \quad \exists f. \top \sqsubseteq C \tag{5.28}$$

The OWL 2 equivalents of assertions (5.27) and (5.28) are respectively (5.29) and (5.30):

```

ObjectProperty: f_inv
ObjectProperty: p1
      ⋮
ObjectProperty: pm
Class: C_f(P1, ..., Pm)
SubClassOf:
  f_inv exactly 1 Thing and
  p1 exactly 1 Thing and
  ⋮
  pm exactly 1 Thing

```

(5.29)

```

ObjectProperty: f_inv
  Domain: C_f(P1, ..., Pm)
  Range: C
ObjectProperty: f
  InverseOf: f_inv

```

(5.30)

For the situation of operation $f()$ (i.e. an operation that takes no parameters and returns no value) the $SR\mathcal{OIQ}^{(D)}$ and OWL 2 assertions respectively are:

$$\exists f^-. \top \sqsubseteq C_{f()} \quad \exists f. \top \sqsubseteq C \tag{5.31}$$

```

ObjectProperty: f_inv
  Domain: C_f
  Range: C
ObjectProperty: f
  InverseOf: f_inv

```

(5.32)

5.4 Translations for Modeller Productivity

This section revisits the translations of enumerations (Section 5.4.1) and binary associations (Section 5.4.2 on p. 74) with the aim to improve modeller efficiency.

5.4.1 Enumerations

The translation of Zedlitz, et. al. assume that enumerations do not have internal structure [118, 119]. The UML specification defines enumerations as data types, but data types in UML class diagrams can also have attributes and operations [64, 104]. Therefore, for the purpose of formal scenario testing, the preference is to translate enumerations to OWL 2 classes (respectively concepts) rather than data types. The reason for this is that it is easier to evolve classes (respectively concepts) than data types to support an internal structure if it becomes apparent from the business requirement that this is required. Hence, the `Colour` enumeration of Figure 2.2 (p. 15) is translated as follows:

```

Class: Colour
  EquivalentTo: {Green, Amber, Red}
Individual: Green
  Types: Colour
Individual: Amber
  Types: Colour
Individual: Red
  Types: Colour
DifferentIndividuals: Green, Amber, Red

```

(5.33)

The translation of the operation is the same as was discussed in Section 5.3 (p. 66). Where an enumeration `Enum` owns attributes `a1, ..., an` that are translated to the data properties `dp1, ..., dpn`, the Easy Key assertion (5.34) has to be added as explained by Zedlitz, et. al. [119]. This assertion enforces that all individuals that have the same values for the data properties `dp1, ..., dpn` represent the same individual.

$$\begin{aligned} \text{Class: Enum} \\ \text{HasKey: dp1, \dots, dpn} \end{aligned} \tag{5.34}$$

The $SRIOQ^{(D)}$ translation is defined in terms of nominals as in (5.35).

$$\text{Colour} \equiv \{\text{Green, Amber, Red}\} \tag{5.35}$$

As explained in Section 5.1.2 (p. 62) Easy Keys cannot be expressed in $SRIOQ^{(D)}$. Hence assertion (5.34) cannot be translated to $SRIOQ^{(D)}$.

5.4.2 Limiting Redundancy of Assertions for Binary Associations

In some instances in the translation of UML class diagram features to OWL 2, Zedlitz, et. al. [118] define assertions that are redundant. That is, some of the explicit assertions can be inferred from already specified assertions. For formal scenario testing the preference is for minimal assertions. For a modeller applying the translation from UML to OWL 2 manually, it is more efficient if the assertions that need to be stated are free of redundancies.

Zedlitz, et. al. [118] define the translation for association A in Figure 2.3 (p. 16) as the assertions (4.14) on p. 53. It states the domain and range restrictions of both object properties \mathbf{a} and $\mathbf{a_inv}$, even though $\mathbf{a_inv}$ is stated to be the inverse of \mathbf{a} and therefore the domain and range of $\mathbf{a_inv}$ can be inferred from \mathbf{a} . Therefore, (4.14) can be rewritten as (5.36).

$$\begin{aligned} \text{ObjectProperty: a} \\ \text{Domain: C} \\ \text{Range: T} \\ \text{ObjectProperty: a_inv} \\ \text{InverseOf: a} \end{aligned} \tag{5.36}$$

For the translation of the association specialization in Figure 2.7(a) (p. 19) Zedlitz, et. al. [118] define the assertions (4.24) on p. 55. Since the assumption is that $\mathbf{a1_inv}$ is the inverse of $\mathbf{a1}$ and $\mathbf{a2_inv}$ is the inverse of $\mathbf{a2}$, it follows that $\mathbf{a2_inv}$ is a sub property of $\mathbf{a1_inv}$ if $\mathbf{a2}$ is a sub property of $\mathbf{a1}$. Hence (4.24) can be rewritten as (5.37).

$$\begin{aligned} \text{ObjectProperty: a2} \\ \text{SubPropertyOf: a1} \end{aligned} \tag{5.37}$$

5.5 Subsetting and Redefinition of Association Ends

In Section 4.6 (p. 54) the translations of association specialization to $SR\mathcal{OIQ}^{(D)}$ and OWL 2 were given. Even though, as explained in Section 2.1.5 (p. 18), subsetting and redefinition of association ends are forms of association specialization, the translations for these were not made explicit. The reason for this is that association specialization is defined at the level of the complete association while subsetting and redefinition are defined at the level of association ends [64].

Subsetting and redefinition (see Figure 2.7(b) and 2.7(c) respectively on p. 19) are both translated to the $SR\mathcal{OIQ}^{(D)}$ and OWL 2 assertions (5.38) and (5.39) respectively.

$$\begin{aligned}
 c_1 &\equiv c_2^- \\
 \exists c_1.\top &\sqsubseteq C_1 \\
 \exists c_1^-. \top &\sqsubseteq C_2 \\
 c_3 &\equiv c_4^- \\
 \exists c_3.\top &\sqsubseteq C_3 \\
 \exists c_3^-. \top &\sqsubseteq C_4 \\
 c_3 &\sqsubseteq c_1 \\
 c_4 &\sqsubseteq c_2
 \end{aligned} \tag{5.38}$$

$$\begin{aligned}
 \text{ObjectProperty: } &c1 \\
 \text{Domain: } &C1 \\
 \text{Range: } &C2 \\
 \text{InverseOf: } &c2 \\
 \text{ObjectProperty: } &c2 \\
 \text{ObjectProperty: } &c3 \\
 \text{Domain: } &C3 \\
 \text{Range: } &C4 \\
 \text{InverseOf: } &c4 \\
 \text{SubPropertyOf: } &c1 \\
 \text{ObjectProperty: } &c4
 \end{aligned} \tag{5.39}$$

5.6 On the Equivalence of Attributes and Binary Associations

In Section 2.1.3 (p. 16) the equivalence of attributes and associations, as stated in the UML specification, was discussed [64]. Here the mathematical equivalence of binary associations and attributes are confirmed.

Consider the attribute $t:T$ of class C as illustrated in Figure 2.1(b) (p. 14). For the translation to $SRDIQ^{(D)}$ an atomic role t is introduced with domain C and range T respectively defined by assertions (5.4) and (5.5) on p. 65. Association A in Figure 2.3 (p. 16) is translated to $SRDIQ^{(D)}$ using the domain assertion (5.6) on p. 65 and the range assertion (5.7) on p. 65. It introduces the atomic role a with domain C and range T . Since the atomic roles t and a correspond with regards to their domains and ranges, it follows that $t \equiv a$ holds. Therefore, an attribute of type T (which can be either a class or a data type) on class C is equivalent to class C having a binary association with class/data type of type T .

Strictly speaking the above result is stronger than what is suggested by the UML specification. The UML specification concedes that attributes and binary associations are equivalent under certain circumstances. However, the above result indicates that attributes and binary associations are equivalent irrespective of the circumstances.

Based on the above result formal scenario testing will make no distinction between associations and attributes. The benefit of this equivalence is that a modeller can decide to use either attributes or associations depending on whether a more compact or a more explicit graphical representation is required.

5.7 A Note on Uniqueness of Names in UML Class Diagrams

Section 2.1.7 (p. 21) explained that all names in UML class diagrams are unique due to the use of qualified names. However, for the purpose of formal scenario testing the uniqueness of names are relaxed. This relaxation affords the opportunity to support the detection of cohesion heuristics that are discussed in Chapter 7 (p. 119).

On the face of it, it may seem as if this section contradicts the efforts made in Section 5.2 (p. 63) to specify various domain and range restrictions as tightly as possible. However, this not the case. Even with relaxing the domain and range restrictions in this section, the resulting domains and ranges of the relevant roles still turn out to be subsets of those specified by Berardi, et. al. [13] in Chapter 4 (p. 49).

This section is structured as follows. Sections 5.7.1 and 5.7.2 (p. 77) respectively discuss the relaxation of qualified names of attributes/associations and operations. When using ontology editors, a related issue that needs to be considered is anonymous classes. How to deal with anonymous classes are discussed in Section 5.7.3 (p. 78).

5.7.1 Attributes and Associations

Figure 2.9 (p. 21) refers in the discussion that follows. In order to stay true to semantics of UML, when attribute a is translated to DL (respectively OWL 2), the translation needs to take cognizance of qualified names. This means that the attribute a for the respective classes $C1$ and $C2$ needs to be translated to two different roles (respectively properties). However,

for the purpose of formal scenario testing, qualified names are intentionally ignored. Hence, the attribute **a** of Figure 2.9 (p. 21) is translated to a single role a (respectively property **a**) of which the domain is the union of the concepts C_1 and C_2 (respectively classes **C1** and **C2**) and the range the union of the concepts T_1 and T_2 (respectively classes **T1** and **T2**). In (5.40) and (5.41) the respective $\mathcal{SROIQ}^{(D)}$ and OWL 2 translations for an attribute **a** belonging to classes **C1** and **C2** are provided.

$$\begin{aligned} \exists a.\top &\sqsubseteq C_1 \sqcup C_2 \\ \exists a^{\bar{}}.\top &\sqsubseteq T_1 \sqcup T_2 \end{aligned} \tag{5.40}$$

$$\begin{aligned} \text{ObjectProperty: } &\mathbf{a} \\ \text{Domain: } &\mathbf{C1} \text{ or } \mathbf{C2} \\ \text{Range: } &\mathbf{T1} \text{ or } \mathbf{T2} \end{aligned} \tag{5.41}$$

5.7.2 Operations and Parameters

In the following discussion Figure 2.10 (p. 21) refers. In terms of qualified names the translation for operations are relaxed in a similar way as for attributes. In line with the translation of operations the atomic concepts $C_{1_{f(P_1, \dots, P_m):R}}$ and $C_{2_{f(P_1, \dots, P_n):R}}$ are introduced for representing the reified relations of the operations **f** appearing in class **C1** and class **C2** respectively. Assuming that $m < n$, the assertions (5.14) and (5.15) for Figure 2.10 are rewritten as (5.42) and (5.43) respectively:

$$\begin{aligned} \exists p_i.\top &\sqsubseteq C_{1_{f(P_1, \dots, P_m):R}} \sqcup C_{2_{f(P_1, \dots, P_n):R}} & \exists p_i^{\bar{}}.\top &\sqsubseteq P_i & i &= 1, \dots, m \\ \exists p_j.\top &\sqsubseteq C_{2_{f(P_1, \dots, P_n):R}} & \exists p_j^{\bar{}}.\top &\sqsubseteq P_j & j &= m+1, \dots, n \end{aligned} \tag{5.42}$$

$$\begin{aligned} \exists f^{\bar{}}.\top &\sqsubseteq C_{1_{f(P_1, \dots, P_m):R}} \sqcup C_{2_{f(P_1, \dots, P_n):R}} & \exists f.\top &\sqsubseteq C_1 \sqcup C_2 \\ \exists r_R.\top &\sqsubseteq C_{1_{f(P_1, \dots, P_m):R}} \sqcup C_{2_{f(P_1, \dots, P_n):R}} & \exists r_R^{\bar{}}.\top &\sqsubseteq R \end{aligned} \tag{5.43}$$

Below the equivalent translations to OWL 2 are provided assuming $\mathbf{k=m+1}$:

$$\begin{array}{l}
\text{ObjectProperty: } p_1 \\
\quad \text{Domain: } C1_f(P_1, \dots, P_m)_R \text{ or } C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } P_1 \\
\quad \vdots \\
\text{ObjectProperty: } p_m \\
\quad \text{Domain: } C1_f(P_1, \dots, P_m)_R \text{ or } C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } P_m \\
\text{ObjectProperty: } p_k \\
\quad \text{Domain: } C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } P_k \\
\quad \vdots \\
\text{ObjectProperty: } p_n \\
\quad C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } P_n
\end{array} \tag{5.44}$$

$$\begin{array}{l}
\text{ObjectProperty: } f_inv \\
\quad \text{Domain: } C1_f(P_1, \dots, P_m)_R \text{ or } C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } C_1 \text{ or } C_2 \\
\text{ObjectProperty: } f \\
\quad \text{InverseOf: } f_inv \\
\text{ObjectProperty: } r_R \\
\quad \text{Domain: } C1_f(P_1, \dots, P_m)_R \text{ or } C2_f(P_1, \dots, P_n)_R \\
\quad \text{Range: } R
\end{array} \tag{5.45}$$

5.7.3 Dealing with Anonymous Classes

In ontology editors any expressions that are not explicitly named are considered to be anonymous classes. The effect is that when an ontology editor reports on inferences, only named classes are displayed. Any anonymous classes which form part of inferences are omitted from the display. This may result in important inferences not being made explicit within the user interface of the ontology editor [99].

To limit problems due to the lack of displaying of anonymous classes, explicit named classes for the unions of classes used in the domains and/or ranges of (5.41), (5.44) and (5.45) are introduced. As an example (5.41) is rewritten as follows:

```

Class: ADomain
    EquivalentTo: C1 or C2
Class: ARange
    EquivalentTo: T1 or T2
ObjectProperty: a
    Domain: ADomain
    Range: ARange

```

(5.46)

In the case where another attribute say *b* exists, which is also shared between classes *C1* and *C2*, another class called *BDomain* equivalent to the union of *C1* and *C2* is introduced. In this case the classes *ADomain* and *BDomain* turn out to be equivalent. Even so, for formal scenario testing the preference is to keep the classes *ADomain* and *BDomain* as separate classes. This decision is motivated by the fact that the domains of attribute *a* and *b* may evolve independently over time. Having the same class representing the domain of both attributes makes such an natural evolution cumbersome.

Lastly, it is worth noting that gathering domains and ranges together as suggested here is not ideal. The most notable downside is that a modeller may forget to gather domains and ranges together, which will make it impossible to detect some of the heuristics discussed in Chapter 7 (p. 119). This problem can be limited and possibly be avoided if tool support can be provided that does the gathering without modeller intervention.

5.8 Why Composition and Aggregation are excluded from Formal Scenario Testing

In this section the reasons are discussed for explicitly excluding aggregation and composition from formal scenario testing. Firstly, aggregation and composition are used to denote whole-part relations [18, 64, 104], but the analysis of for instance Henderson-Sellers, et. al. [53], Barbier, et. al. [9], and Keet [69] have respectively pointed out various aspects with regards to aggregation and composition in which the UML specification is under specified. All have proposed means for addressing the shortcomings of the UML specification, but to date these changes have not been incorporated into the UML specification [64].

Secondly, Wazlawick [116] observes that the real advantage of composition and aggregation is that the attributes of the parts are often used to derive attributes of the whole. As an example he mentions that the total value of an order (whole) is derived of the value of each of its items (parts). However, this to him is a design concern rather than a conceptual modelling concern. From a conceptual modelling perspective, he believes that modellers often apply aggregation and composition inappropriately (that is, where whole-part relations are not

present) and that their use seldom have real benefit. Hence he suggests avoiding or even abolishing their use.

Thirdly, since the goal of this dissertation is to be able to reason on UML class and object diagrams, it is needed to be able to provide a concise translation of whole-part relations to DLs. However, reasoning on whole-part relations is a well-known challenge in DLs [15, 71, 108].

5.9 Contribution and Related Research

In this section the contributions of this chapter and related research are discussed.

5.9.1 Contribution

The emphasis of the research contribution of this chapter is on the translation of UML class diagrams and UML object diagrams to $SR\mathcal{OIQ}^{(D)}$ and OWL 2 for the purpose of formal scenario testing. Below the contributions of this chapter are briefly reviewed:

1. A means for translating identity constraints on UML class diagrams to OWL 2 is suggested which can be used to validate identity constraints on instances in a UML object diagram (Section 5.1 on p. 61).
2. The domains and ranges of the roles introduced for the translation of attributes/associations and operations have been tightened in order to accurately represent the semantics of these UML features (see Section 5.2 on p. 63).
3. The main contribution of Section 5.3 (p. 66) is to suggest, based on the existing translation of operations to $ALCQI$, translations of operations to $SR\mathcal{OIQ}^{(D)}$ and OWL 2. Fine grained contributions in this regard are:
 - (a) For the purpose of formal scenario testing the existing translation of operations are refined to enable the detection of various modelling heuristics. Therefore preference is given to an explicit role name convention rather than arbitrary role names (Section 5.3.1 on p. 67).
 - (b) A class that has a particular operation defined on it, necessarily must be able to call the operation. This requirement is enforced through the addition of relevant assertions for $SR\mathcal{OIQ}^{(D)}$ and OWL 2 in Section 5.3.2 (p. 68).
 - (c) Existing translations of operations handle the translation of operations with no parameters and operations with one or more parameters differently. Section 5.3.3 (p. 69) modifies the translation of operations with no parameters to be similar to that of operations with one or more parameters.
 - (d) Section 5.3.4 (p. 69) provides a translation of operations to OWL 2.

- (e) In order to ensure that the return value of an operation called on an instance of a class is deterministic for a given set of parameters, an identity constraint needs to be applied to $\mathcal{SROIQ}^{(D)}$ and OWL 2 (Section 5.3.5 on p. 71).
 - (f) The UML specification allows operations without return values. The translation of operations without return values is made explicit in Section 5.3.6 (p. 71).
4. The $\mathcal{SROIQ}^{(D)}$ and OWL 2 translations for subsetting and redefinition of association ends have been made explicit in Section 5.5 (p. 75).
 5. Some of the translations of UML class diagram features to OWL 2 are not efficient in the context of formal scenario testing. Therefore Section 5.4.1 (p. 73) defines an alternative translation for enumerations, while Section 5.4.2 (p. 74) addresses redundancies regarding the translations of associations and association specializations.
 6. The UML specification states that attributes and binary associations are equivalent under certain circumstances. Section 5.6 (p. 75) confirms this intuition by providing a mathematical proof that these are equivalent irrespective of the circumstances.
 7. Names in UML class diagrams are unique and hence any formalization of UML class diagrams has to take this into account. Section 5.7 (p. 76) explains how this is handled for formal scenario testing in order to enable detection of various modelling heuristics.
 8. Composition and aggregation are features of UML class diagrams that are explicitly excluded from formal scenario testing. The rationale for this decision is given in Section 5.8 (p. 79).

5.9.2 Related Research

Here only related research with regards to the translation of UML class diagrams to DLs and OWL 2 is discussed. Related research with regards to formal scenario testing is discussed in Section 6.5.2 (p. 117).

Identity constraints have been applied to UML class diagrams by Cali, et. al. [24] and Berardi, et. al [13] using the DLs \mathcal{DLR}_{ifd} and \mathcal{ALCQI} . \mathcal{DLR}_{ifd} has specific constructors which can be used to express functional dependency and identity constraints. However, it does not have constructors for data types [24, 13]. The DL \mathcal{ALCQI} does not have a specific constructor for applying identity constraints. Rather, identity constraints are implicit in \mathcal{ALCQI} due to it having the tree-model property. In addition, \mathcal{ALCQI} does not have support for data types [13].

Queralt, et. al. [100] defined OCL-Lite which is a decidable subset of OCL. OCL provides a means through which additional constraints can be defined on classes in a UML class diagram which cannot otherwise be expressed. OCL-Lite can be translated to the DL \mathcal{ALCT}

which is a DL which is strictly less expressive than the DLs $ALCQI$ and $SRQIQ^{(D)}$. Hence, theoretically at least the capabilities of OCL-Lite must be available in $SRQIQ^{(D)}$ and OWL 2. The only reason for explicitly excluding OCL constraints in this dissertation is due to scope constraints.

5.10 Summary

This chapter concludes with a summary of the translations of UML class diagram features to $SRQIQ^{(D)}$ and OWL 2 in Table 5.1. For completeness sake the translations of all UML class diagram features, as have been discussed in Chapter 2 (p. 13), are repeated here even in the case where the translation does not differ from that given in Table 4.1 (p. 59).

Table 5.1: UML class diagram translation to $SRQIQ^{(D)}$ and OWL 2


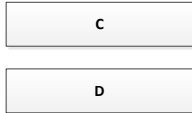
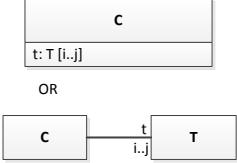
UML class diagram feature	$SRQIQ^{(D)}$	OWL 2	Ref.
 <p>Class C</p>	C	Class: C	4.1 p. 49
 <p>Classes C and D</p>	$C \sqsubseteq \neg D$	Class: C Class: D DisjointWith: C	4.1 p. 49
 <p>Attribute t of type T with multiplicity [i..j] OR Association end t of type T with multiplicity [i..j]</p>	$\exists t.T \sqsubseteq C$ $\exists t^-.T \sqsubseteq T$ $C \sqsubseteq (\geq i t.T) \sqcap (\leq j t.T)$	Class: C SubClassOf: (t min i Thing) and (t max j Thing) Class: T ObjectProperty: t Domain: C Range: T	4.2.1 p. 51, 5.2.1 p. 64 and 5.6 p. 75

Table 5.1: UML class diagram translation to $SR_{OIQ}^{(D)}$ and OWL 2

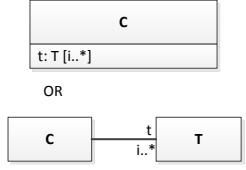
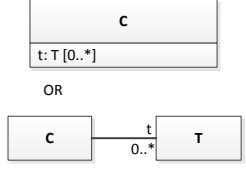
UML class diagram feature	$SR_{OIQ}^{(D)}$	OWL 2	Ref.
 <p>Attribute t of type T with multiplicity $[i..*]$</p> <p>OR</p> <p>Association end t of type T with multiplicity $[i..*]$</p>	$\exists t.T \sqsubseteq C$ $\exists t^-.T \sqsubseteq T$ $C \sqsubseteq (\geq i t.T)$	<pre> Class: C SubClassOf: (t min i Thing) Class: T ObjectProperty: t Domain: C Range: T </pre>	<p>4.2.1 p. 51, 5.2.1 p. 64 and 5.6 p. 75</p>
 <p>Attribute t of type T with multiplicity $[0..*]$</p> <p>OR</p> <p>Association end t of type T with multiplicity $[0..*]$</p>	$\exists t.T \sqsubseteq C$ $\exists t^-.T \sqsubseteq T$	<pre> Class: C Class: T ObjectProperty: t Domain: C Range: T </pre>	<p>4.2.1 p. 51, 5.2.1 p. 64 and 5.6 p. 75</p>

Table 5.1: UML class diagram translation to $SROIQ^{(D)}$ and OWL 2

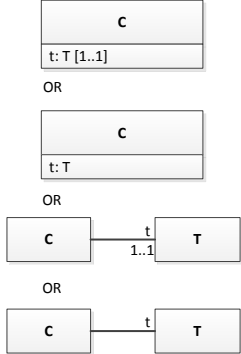
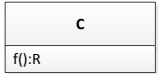
UML class diagram feature	$SROIQ^{(D)}$	OWL 2	Ref.
 <p>Attribute t of type T with multiplicity $[1..1]$</p> <p>OR</p> <p>Association end t of type T with multiplicity $[1..1]$</p>	$\exists t.T \sqsubseteq C$ $\exists t^-.T \sqsubseteq T$ $C \sqsubseteq \exists t.T \sqcap (\leq 1 t.T)$	<pre> Class: C SubClassOf: t exactly 1 Thing Class: T </pre>	<p>4.2.1 p. 51, 5.2.1 p. 64 and 5.6 p. 75</p>
 <p>Operation f with no parameters returning a value of type R</p>	$C_{f():R} \sqsubseteq \exists f^-.T \sqcap (\leq 1 f^-.T) \sqcap \exists r_R.T \sqcap (\leq 1 r_R.T)$ $\exists f^-.T \sqsubseteq C_{f():R}$ $\exists f.T \sqsubseteq C$ $\exists r_R.T \sqsubseteq C_{f():R}$ $\exists r_R^-.T \sqsubseteq R$ $C \sqsubseteq \exists f.C_{f():R}$ <p>Note that determinism of the return value cannot be enforced since rule-like constructs are required which are not supported in $SROIQ^{(D)}$</p>	<pre> Class: C_f()_R SubClassOf: f_inv exactly 1 Thing and r_R exactly 1 Thing HasKey: f_inv ObjectProperty: f_inv Domain: C_f()_R Range: C ObjectProperty: f InverseOf: f_inv ObjectProperty: r_R Domain: C_f()_R Range: R Class: C SubClassOf: f some C_f()_R Class: R </pre>	<p>5.2.3 p. 65 and 5.3 p. 66</p>

Table 5.1: UML class diagram translation to $SR\mathcal{OIQ}^{(D)}$ and OWL 2

UML class diagram feature	$SR\mathcal{OIQ}^{(D)}$	OWL 2	Ref.
<p>Operation f with no parameters and no return value</p>	$C_{f()} \sqsubseteq \exists f^-. \top \sqcap (\leq 1 f^-. \top)$ $\exists f^-. \top \sqsubseteq C_{f()}$ $\exists f. \top \sqsubseteq C$ $C \sqsubseteq \exists f. C_{f()}$	<pre> Class: C_f() SubClassOf: f_inv exactly 1 Thing ObjectProperty: f_inv Domain: C_f() Range: C ObjectProperty: f InverseOf: f_inv Class: C SubClassOf: f some C_f() </pre>	5.2.3 p. 65 and 5.3 p. 66
<p>Operation f with parameters p_1, \dots, p_m respectively of type P_1, \dots, P_m returning a value of type R</p>	$C_{f(P_1, \dots, P_m):R} \sqsubseteq$ $\exists f^-. \top \sqcap (\leq 1 f^-. \top) \sqcap$ $\exists p_1. \top \sqcap (\leq 1 p_1. \top) \sqcap$ \vdots $\exists p_m. \top \sqcap (\leq 1 p_m. \top) \sqcap$ $\exists r_R. \top \sqcap (\leq 1 r_R. \top)$ $\exists p_i. \top \sqsubseteq C_{f(P_1, \dots, P_m):R}$ $\exists p_i^-. \top \sqsubseteq P_i \quad i = 1, \dots, m$ $\exists f^-. \top \sqsubseteq C_{f(P_1, \dots, P_m):R}$ $\exists f. \top \sqsubseteq C$ $\exists r_R. \top \sqsubseteq C_{f(P_1, \dots, P_m):R}$ $\exists r_R^-. \top \sqsubseteq R$ $C \sqsubseteq \exists f. C_{f(P_1, \dots, P_m):R}$ <p>Note that determinism of the return value cannot be enforced since rule-like constructs are required which are not supported in $SR\mathcal{OIQ}^{(D)}$</p>	<pre> Class: P1 : : Class: Pm Class: C_f(P1, ..., Pm)_R SubClassOf: f_inv exactly 1 Thing and p1 exactly 1 Thing and : pm exactly 1 Thing and r_R exactly 1 Thing HasKey: f_inv, p1, ..., pm ObjectProperty: p1 Domain: C_f(P1, ..., Pm)_R Range: P1 : : ObjectProperty: pm Domain: C_f(P1, ..., Pm)_R Range: Pm ObjectProperty: f_inv Domain: C_f(P1, ..., Pm)_R Range: C ObjectProperty: f InverseOf: f_inv ObjectProperty: r_R Domain: C_f(P1, ..., Pm)_R Range: R Class: C SubClassOf: f some C_f(P1, ..., Pm)_R Class: R </pre>	5.2.3 p. 65 and 5.3 p. 66

Table 5.1: UML class diagram translation to $SR\mathcal{OIQ}^{(D)}$ and OWL 2

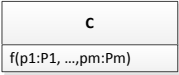
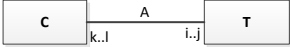
UML class diagram feature	$SR\mathcal{OIQ}^{(D)}$	OWL 2	Ref.
 <p>Operation f with parameters p_1, \dots, p_m respectively of type P_1, \dots, P_m and no return value</p>	$C_{f(P_1, \dots, P_m)} \sqsubseteq$ $\exists f^-.T \sqcap (\leq 1 f^-.T) \sqcap$ $\exists p_1.T \sqcap (\leq 1 p_1.T) \sqcap$ \vdots $\exists p_m.T \sqcap (\leq 1 p_m.T)$ $\exists p_i.T \sqsubseteq C_{f(P_1, \dots, P_m)}$ $\exists p_i^-.T \sqsubseteq P_i \quad i = 1, \dots, m$ $\exists f^-.T \sqsubseteq C_{f(P_1, \dots, P_m)}$ $\exists f.T \sqsubseteq C$ $C \sqsubseteq \exists f.C_{f(P_1, \dots, P_m)}$	<pre> Class: P1 : : Class: Pm Class: C_f(P1, ..., Pm) SubClassOf: f_inv exactly 1 Thing and p1 exactly 1 Thing and : : pm exactly 1 Thing ObjectProperty: p1 Domain: C_f(P1, ..., Pm) Range: P1 : : ObjectProperty: pm Domain: C_f(P1, ..., Pm) Range: Pm ObjectProperty: f_inv Domain: C_f(P1, ..., Pm) Range: C ObjectProperty: f InverseOf: f_inv Class: C SubClassOf: f some C_f(P1, ..., Pm) </pre>	<p>5.2.3 p. 65 and 5.3 p. 66</p>
 <p>Association A exists between classes C and T</p>	$\exists a.T \sqsubseteq C$ $\exists a^-.T \sqsubseteq T$ $C \sqsubseteq (\geq i a.T) \sqcap (\leq j a.T)$ $T \sqsubseteq (\geq k a^-.T) \sqcap (\leq l a^-.T)$	<pre> ObjectProperty: a Domain: C Range: T ObjectProperty: a_inv InverseOf: a Class: C SubClassOf: (a min i Thing) and (a max j Thing) Class: T SubClassOf: (a_inv min k Thing) and (a_inv max l Thing) </pre>	<p>4.4 p. 53, 5.2 p. 63 and 5.4.2 p. 74</p>

Table 5.1: UML class diagram translation to $SRQIQ^{(D)}$ and OWL 2

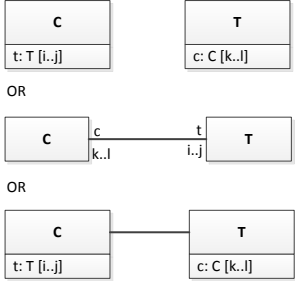
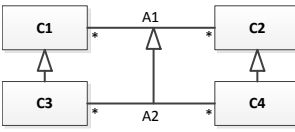
UML class diagram feature	$SRQIQ^{(D)}$	OWL 2	Ref.
 <p>Attribute t of type T with multiplicity [i..j] belongs to class C and attribute c of type C with multiplicity [k..l] belongs to class T</p> <p>OR</p> <p>Association end t with multiplicity [i..j] is associated with class C and association end c with multiplicity [k..l] is associated with class T</p> <p>OR</p> <p>Line notation is used to make explicit that attributes t and c are also association ends</p>	$\exists t.T \sqsubseteq C$ $\exists t^{-}.T \sqsubseteq T$ $C \sqsubseteq (\geq i t.T) \sqcap (\leq j t.T)$ $c \equiv t^{-}$ $T \sqsubseteq (\geq k c.T) \sqcap (\leq l c.T)$	<pre> ObjectProperty: t Domain: C Range: T Class: C SubClassOf: (t min i Thing) and (t max j Thing) ObjectProperty: c InverseOf: t Class: T SubClassOf: (c min k Thing) and (c max l Thing) </pre>	<p>4.4 p. 53, 5.2 p. 63 and 5.6 p. 75</p>
 <p>Association A2 specializes association A1</p>	$\exists a_1.T \sqsubseteq C_1$ $\exists a_1^{-}.T \sqsubseteq C_2$ $\exists a_2.T \sqsubseteq C_3$ $\exists a_2^{-}.T \sqsubseteq C_4$ $a_2 \sqsubseteq a_1$	<pre> Class: C1 Class: C2 Class: C3 Class: C4 ObjectProperty: a1 Domain: C1 Range: C2 ObjectProperty: a2 Domain: C3 Range: C4 SubPropertyOf: a1 </pre>	<p>4.6 p. 54 and 5.4.2 p. 74</p>

Table 5.1: UML class diagram translation to $SRQIQ^{(D)}$ and OWL 2

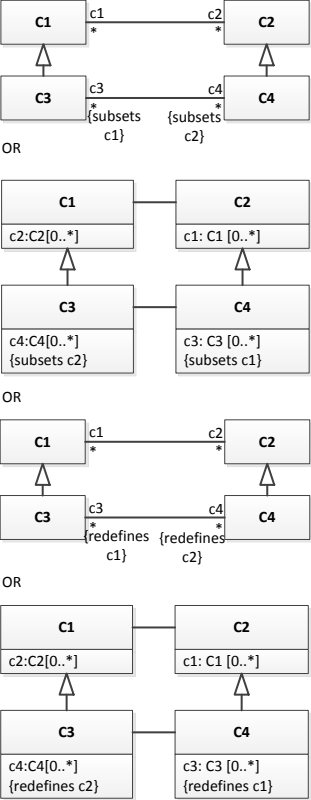
UML class diagram feature	$SRQIQ^{(D)}$	OWL 2	Ref.
 <p>Association end (resp. attribute) c3 subsets association end (resp. attribute) c1 and association end (resp. attribute) c4 subsets association end (resp. attribute) c2</p> <p>OR</p> <p>Association end (resp. attribute) c3 redefines association end (resp. attribute) c1 and association end (resp. attribute) c4 redefines association end (resp. attribute) c2</p>	$c_1 \equiv c_2^-$ $\exists c_1. \top \sqsubseteq C_1$ $\exists c_1^-. \top \sqsubseteq C_2$ $c_3 \equiv c_4^-$ $\exists c_3. \top \sqsubseteq C_3$ $\exists c_3^-. \top \sqsubseteq C_4$ $c_3 \sqsubseteq c_1$ $c_4 \sqsubseteq c_2$	<p>Class: C1 Class: C2 Class: C3 Class: C4 ObjectProperty: c1 Domain: C1 Range: C2 InverseOf: c2 ObjectProperty: c2 ObjectProperty: c3 Domain: C3 Range: C4 InverseOf: c4 SubPropertyOf: c1 ObjectProperty: c4</p>	<p>5.5 p. 75</p>

Table 5.1: UML class diagram translation to $SR_{OIQ}^{(D)}$ and OWL 2

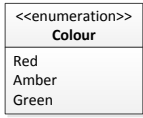
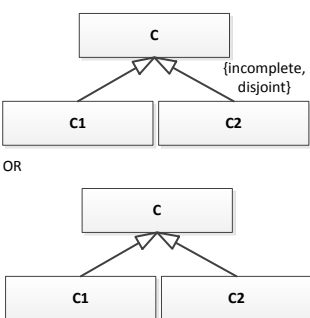
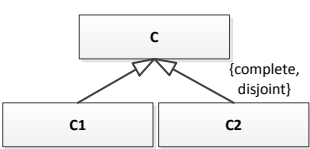
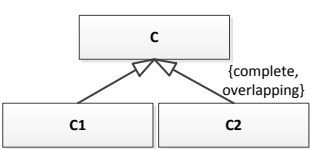
UML class diagram feature	$SR_{OIQ}^{(D)}$	OWL 2	Ref.
 <p>The Colour enumeration consists of the colours Red, Amber and Green</p>	$Colour \equiv \{Green, Amber, Red\}$ $Green \not\approx Amber$ $Green \not\approx Red$ $Amber \not\approx Red$	Class: Colour EquivalentTo: {Green, Amber, Red} Individual: Green Types: Colour Individual: Amber Types: Colour Individual: Red Types: Colour DifferentIndividuals: Green, Amber, Red	5.4.1 p. 73
 <p>Class C is specialized by the disjoint classes C1 and C2 which do not cover class C</p>	$C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$ $C_1 \sqsubseteq \neg C_2$	Class: C Class: C1 SubClassOf: C Class: C2 SubClassOf: C DisjointClasses: C1, C2	4.5 p. 54
 <p>Class C is specialized by the disjoint classes C1 and C2 which cover class C</p>	$C \sqsubseteq C_1 \sqcup C_2$ $C_1 \sqsubseteq \neg C_2$ $C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	Class: C DisjointUnionOf: C1, C2 Class: C1 SubClassOf: C Class: C2 SubClassOf: C	4.5 p. 54
 <p>Class C is specialized by the overlapping classes C1 and C2 which cover class C</p>	$C \sqsubseteq C_1 \sqcup C_2$ $C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	Class: C SubClassOf: C1 or C2 Class: C1 SubClassOf: C Class: C2 SubClassOf: C	4.5 p. 54

Table 5.1: UML class diagram translation to $SRQIQ^{(D)}$ and OWL 2

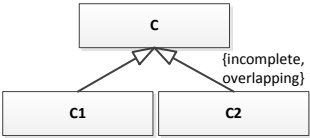
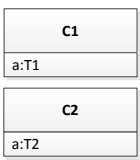
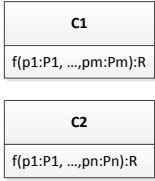
UML class diagram feature	$SRQIQ^{(D)}$	OWL 2	Ref.
 <p>Class C is specialized by the overlapping classes C1 and C2 which do not cover class C</p>	$C_1 \sqsubseteq C$ $C_2 \sqsubseteq C$	<pre>Class: C Class: C1 SubClassOf: C Class: C2 SubClassOf: C</pre>	4.5 p. 54
 <p>Classes C1 and C2 both have an attribute a respectively of type T1 and T2.</p>	$\exists a.T \sqsubseteq C_1 \sqcup C_2$ $\exists a^-.T \sqsubseteq T_1 \sqcup T_2$ $C_1 \sqsubseteq \exists a.T \sqcap (\leq 1 a.T)$ $C_2 \sqsubseteq \exists a.T \sqcap (\leq 1 a.T)$	<pre>Class: C1 SubClassOf: t exactly 1 Thing Class: C2 SubClassOf: t exactly 1 Thing Class: T1 Class: T2 ObjectProperty: a Domain: C1 or C2 Range: T1 or T2</pre>	4.2.1 p. 51 and 5.2.1 p. 64 and 5.7.1 p. 76

Table 5.1: UML class diagram translation to $SR\mathcal{OIQ}^{(D)}$ and OWL 2

UML class diagram feature	$SR\mathcal{OIQ}^{(D)}$	OWL 2	Ref.
 <p>Classes C1 and C2 both have an operation f with return value R and respective parameters $p1:P1, \dots, pm:Pm$ and $p1:P1, \dots, pn:Pn$ with $m < n$.</p>	$C_{1f(P_1, \dots, P_m):R} \sqsubseteq$ $\exists f^-.T \sqcap (\leq 1f^-.T) \sqcap$ $\exists p_1.T \sqcap (\leq 1p_1.T) \sqcap$ \vdots $\exists p_m.T \sqcap (\leq 1p_m.T) \sqcap$ $\exists r_R.T \sqcap (\leq 1r_R.T)$ $C_1 \sqsubseteq \exists f.C_{1f(P_1, \dots, P_m):R}$ $C_{2f(P_1, \dots, P_n):R} \sqsubseteq$ $\exists f^-.T \sqcap (\leq 1f^-.T) \sqcap$ $\exists p_1.T \sqcap (\leq 1p_1.T) \sqcap$ \vdots $\exists p_n.T \sqcap (\leq 1p_n.T) \sqcap$ $\exists r_R.T \sqcap (\leq 1r_R.T)$ $C_2 \sqsubseteq \exists f.C_{2f(P_1, \dots, P_n):R}$ $\exists p_i.T \sqsubseteq C_{1f(P_1, \dots, P_m):R}$ $\sqcup C_{2f(P_1, \dots, P_n):R}$ $\exists p_i^-.T \sqsubseteq P_i \quad i = 1, \dots, m$ $\exists p_j.T \sqsubseteq C_{2f(P_1, \dots, P_n):R}$ $\exists p_j^-.T \sqsubseteq P_j \quad j = m + 1, \dots, n$ $\exists f^-.T \sqsubseteq C_{1f(P_1, \dots, P_m):R}$ $\sqcup C_{2f(P_1, \dots, P_n):R}$ $\exists f.T \sqsubseteq C_1 \sqcup C_2$ $\exists r_R.T \sqsubseteq C_{1f(P_1, \dots, P_m):R}$ $\sqcup C_{2f(P_1, \dots, P_n):R}$ $\exists r_R^-.T \sqsubseteq R$ <p>Note that determinism of the return values is not enforced.</p>	Class: P1 \vdots Class: Pn Class: R Class: C1_f(P1, ..., Pm)_R SubClassOf: f_inv exactly 1 Thing and p1 exactly 1 Thing and \vdots pm exactly 1 Thing and r_R exactly 1 Thing HasKey: f_inv, p1, ..., pm Class: C1 SubClassOf: f some C1_f(P1, ..., Pm)_R Class: C2_f(P1, ..., Pn)_R SubClassOf: f_inv exactly 1 Thing and p1 exactly 1 Thing and \vdots pn exactly 1 Thing and r_R exactly 1 Thing HasKey: f_inv, p1, ..., pn Class: C2 SubClassOf: f some C2_f(P1, ..., Pn)_R ObjectProperty: p1 Domain: C1_f(P1, ..., Pm)_R or C2_f(P1, ..., Pn)_R Range: P1 \vdots ObjectProperty: pm Domain: C1_f(P1, ..., Pm)_R or C2_f(P1, ..., Pn)_R Range: Pm ObjectProperty: pk Domain: C2_f(P1, ..., Pn)_R Range: Pk \vdots ObjectProperty: pn C2_f(P1, ..., Pn)_R Range: Pn ObjectProperty: f_inv Domain: C1_f(P1, ..., Pm)_R or C2_f(P1, ..., Pn)_R Range: C1 or C2 ObjectProperty: f InverseOf: f_inv ObjectProperty: r_R Domain: C1_f(P1, ..., Pm)_R or C2_f(P1, ..., Pn)_R Range: R	5.2.3 p. 65 and 5.3 p. 66 and 5.7.2 p. 77

Chapter 6

Formal Scenario Testing

This chapter introduces formal scenario testing, which consists of an approach, techniques and guidelines. The approach details the steps that need be followed to apply formal scenario testing, the techniques define the kind of scenario tests that can be constructed and the guidelines direct the creation and structuring of scenario tests.

Formal scenario testing represents the main contribution of this dissertation. The intent of the translations of UML class diagram features to OWL 2, as was discussed in Chapter 5 (p. 60), is to support formal scenario testing. In Chapter 7 (p. 119) formal scenario testing is used to detect cohesion heuristic violations and validate adherence to cohesion heuristics. How chapters 5–7 relate to each other as is depicted in Figure 6.1.

This chapter is organized as follows: Section 6.1 details the formal scenario testing approach, Sections 6.2 (p. 94) and 6.3 (p. 103) respectively define the related techniques and guidelines and Section 6.4 (p. 108) discusses a small case study where formal scenario testing was applied on a real-world project.

6.1 Approach

Formal scenario testing pertains specifically to the situation where at least one UML class diagram is created as part of the requirements specification process. Formal scenario testing is envisaged as a means for validating that such a UML class diagram is an accurate representation

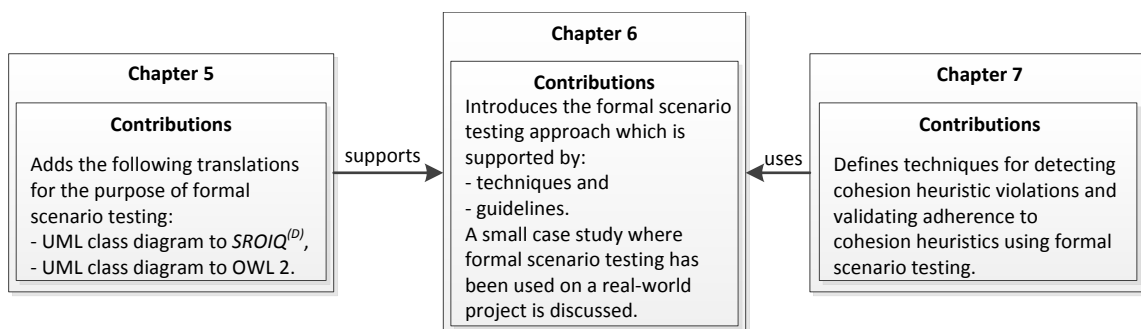


Figure 6.1: Relation of contributions of chapters 5–7.

of the requirements of the business domain. In order to validate a UML class diagram using formal scenario testing, the modeller will need to follow a number of steps, which are detailed in Section 6.1.1. Key characteristics of the formal scenario testing approach are discussed in Section 6.1.2 (p. 93).

6.1.1 Steps of the Formal Scenario Testing Approach

The steps for applying formal scenario testing are as follows:

1. Based on the requirements gathered as part of the requirements elicitation process, the modeller creates a UML class diagram. This UML class diagram represents the key concepts, along with the relations between concepts (conceptual schema), of the business domain.
2. The UML class diagram is translated to OWL 2 and checked for consistency using an ontology editor such as for instance Protégé. If any inconsistencies are found, the modeller must go back to step 1 and correct the UML class diagram. If no inconsistencies are found (that is, the UML class diagram is consistent), the modeller can continue to step 3.
3. Since a consistent UML class diagram does not guarantee that it represents the business requirements accurately, various scenario tests can be applied using the formal scenario testing techniques, which are introduced in Section 6.2 (p. 94). A scenario test represents specific instances of the classes in the UML class diagram, which can be represented using a UML object diagram. The UML object diagram is translated to OWL 2, which enables reasoning on the UML object diagram. If running the reasoner on the OWL 2 translation of a scenario test indicates that the scenario test has an unexpected result (based on the formal scenario testing technique that is applied), the UML class diagram needs to be remedied and therefore the modeller needs to go back to step 1.
4. Step 3 is repeated until there are no further scenario tests to apply. This completes the formal scenario testing process.

Formal scenario testing has a number of key characteristics that are discussed in the next section.

6.1.2 Key Characteristics of Formal Scenario Testing

In this section some key characteristics of formal scenario testing are discussed. Firstly, as mentioned in Section 1.4.2 (p. 9), formal scenario testing differs from typical DL formalizations of UML class diagrams in that it is concerned with validation rather than verification.

Secondly, existing research has focused on checking consistency of a UML class diagram (respectively in DLs the TBox¹). Formal scenario testing extends existing research by checking consistency of scenario tests expressed as UML object diagrams (respectively the ABox in DLs) for the UML class diagram that was created in step 1. In applying formal scenario testing, the actual UML object diagram is seldomly drawn, except for illustration purposes. Rather, scenario tests are often created directly in OWL 2.

Thirdly, it is important to note that the benefits gained from formal scenario testing is directly proportional to the quality and extend of scenario tests considered in step 3. If only a handful of scenario tests is considered with little business impact, the potential benefits of formal scenario testing will be limited.

Lastly, the UML class diagram in step 1 can be created incrementally over time. Even though formal scenario testing can be applied to a UML class diagram of the complete business domain, a complete UML class diagram is not a prerequisite for applying formal scenario testing. Indeed, where formal scenario testing was applied on a real-world project, it was found that creating the UML class diagram incrementally is more efficient. This is discussed in Section 6.4.4 (p. 114).

6.2 Techniques

Three techniques are defined for constructing scenario tests for the purpose of formal scenario testing namely *consistent scenario tests*, *inconsistent scenario tests* and *classification scenario tests*. These techniques are discussed in Sections 6.2.1–6.2.3. Due to the particular techniques applied in formal scenario testing, existing justification reasoning services are not an exact fit for formal scenario testing. Therefore Section 6.2.4 (p. 102) revisits justifications in the context of formal scenario testing and explains how abduction may be better suited.

6.2.1 Consistent Scenario Tests

A scenario that is allowed in a particular business context can be tested for consistency. As an example, consider a UML class diagram that models a robot (traffic light) as illustrated in Figure 6.2. Each of the colours of the robot is modelled to be of type `boolean` and hence each of the colours can be set to `true` or `false` depending on the colour of the robot that is represented. In listing (6.1) the equivalent OWL 2 translation for the UML class diagram in Figure 6.2 is given. In accordance with step 2 in Section 6.1.1 (p. 93) the reasoner is run on listing (6.1) to verify that the UML class diagram is consistent.

¹ Strictly speaking, a UML class diagram corresponds to a TBox and possibly an RBox. All DL translations of UML class diagrams have a TBox, but since not all DL translations of UML class diagrams necessarily have an RBox, the convention used here will be to refer to a TBox with the understanding that the inclusion of an RBox is implied where relevant.

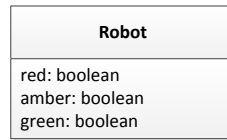


Figure 6.2: A UML class representing a robot.

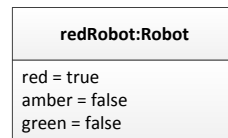


Figure 6.3: An instance representing a red robot.

A scenario that is permissible for the robot example is when the robot is red. This scenario is represented in the UML object diagram of Figure 6.3 with the equivalent scenario represented in OWL 2 in (6.2). If the reasoner is run on the ontology represented by listings (6.1) and (6.2), it confirms that the ontology is consistent. Since this scenario represents an allowed scenario, the formal scenario testing approach expects this ontology to be consistent. If the ontology turned out to be inconsistent, it would represent an unexpected result. Setting up a scenario and running the reasoner to validate the scenario agrees with step 3 of the formal scenario testing process.

```

DataProperty: red
  Domain: Robot
  Range: boolean
DataProperty: amber
  Domain: Robot
  Range: boolean
DataProperty: green
  Domain: Robot
  Range: boolean
Class: Robot
SubClassOf:
  red exactly 1 boolean,
  amber exactly 1 boolean,
  green exactly 1 boolean
  
```

(6.1)

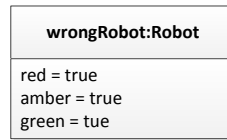


Figure 6.4: An instance of Robot that should be disallowed.

```

Individual: redRobot
Types: Robot
Facts:
    red true,
    amber false,
    green false

```

(6.2)

In terms of DLs, the UML class diagram (Figure 6.2) and the OWL 2 translation representing a robot (listing (6.1)), both correspond with the TBox. Similarly the UML object diagram (Figure 6.3) and the OWL 2 translation representing the scenario (listing 6.2), correspond with the ABox.

6.2.2 Inconsistent Scenario Tests

The corresponding ontology of a scenario that should be disallowed, based on the business requirements, is expected to be inconsistent. In the discussion that follows the robot example of Figure 6.2 (p. 95) refers.

A scenario that should be disallowed for a robot is where all the colours are active simultaneously. Accordingly, for the `wrongRobot` instance the attributes `red`, `amber` and `green` are all set to `true` as shown in Figure 6.4. If listing (6.1) (TBox) and the translation of Figure 6.4 to OWL 2 (ABox) is used, and the reasoner is run on this ontology, the ontology is consistent. Since, from a formal scenario testing perspective, this is a scenario that is disallowed, a consistent ontology represents an unexpected result. Therefore, in accordance with step 3 on p. 93 of the formal scenario testing approach, for an unexpected result the modeller needs to go back to step 1 (p. 93). Hence, the modeller needs to redesign the UML class diagram such that the disallowed scenario test will result in an inconsistent ontology when it is translated to OWL 2.

In Figure 6.5 the redesigned UML class diagram is given with the associated OWL 2 translation provided in listing (6.3). If the scenario where all the colours of the robot are active (see listing (6.4)) is considered again, the resulting ontology is now inconsistent as expected. The partial list of explanations for the inconsistency is shown in Figure 6.6 (p. 98).

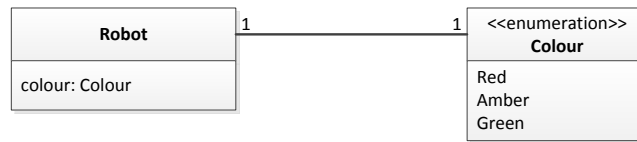


Figure 6.5: A redesign of the Robot class.

```

Individual: Red
  Types: Colour
Individual: Amber
  Types: Colour
Individual: Green
  Types: Colour
DifferentIndividuals: Red, Amber, Green
Class: Colour
  EquivalentTo: {Red, Amber, Green}
ObjectProperty: colour
  Domain: Robot
  Range: Colour
Class: Robot
  SubClassOf: colour exactly 1 Thing
  
```

(6.3)

```

Individual: brokenRobot
  Facts:
    colour Red,
    colour Amber,
    colour Green
  
```

(6.4)

6.2.3 Classification Scenario Tests

Business often has different products on offer with each product having different features. If two products have the same features, it usually indicates that one of the products is redundant. Redundancies can be identified by applying classification to the OWL 2 translation of object instances with the appropriate associated features. As an example, consider the case where a UML class diagram mistakenly contains both a `Robot` and a `TrafficLight` class with the same attributes as illustrated in Figure 6.7 (p. 98).

The translation of Figure 6.7 to OWL 2 is given in listing (6.5) on p. 99. The classes `ColourDomain` and `LastMaintenanceDateDomain` are introduced to represent the domains of the properties `colour` and `lastMaintenanceDate` to cater for anonymous classes in ontology

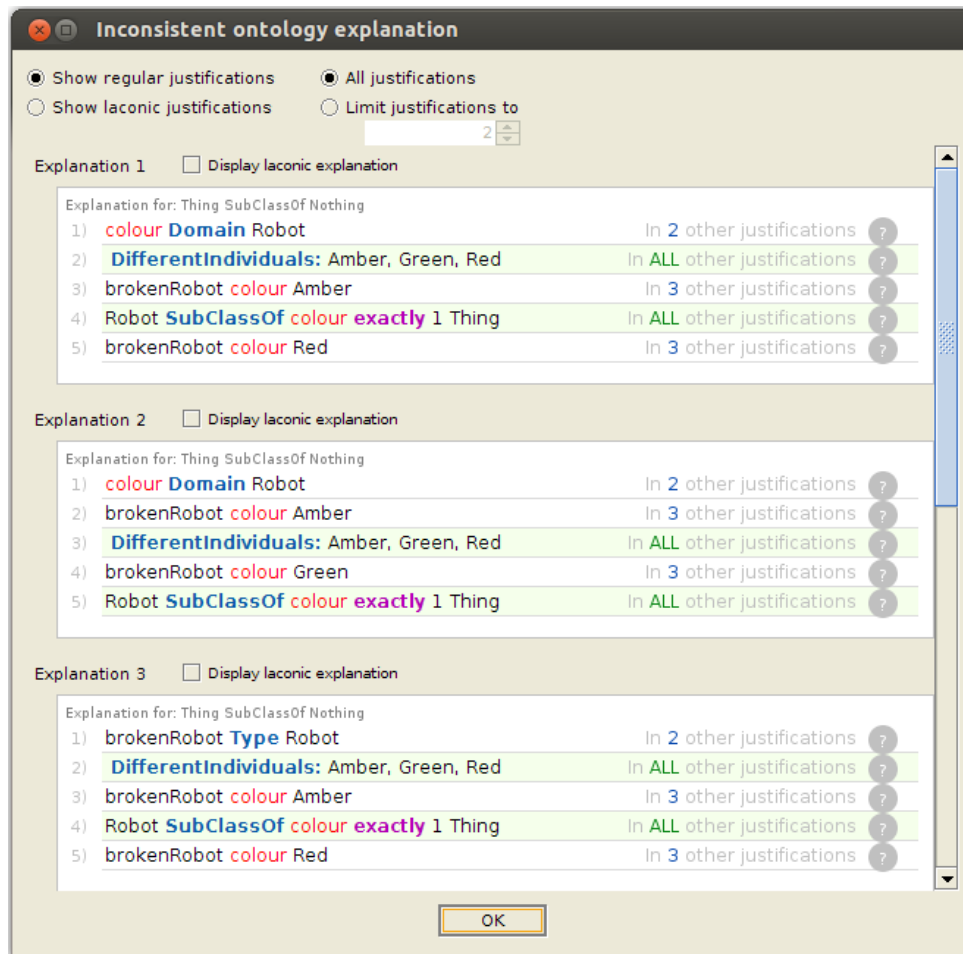


Figure 6.6: Explanations for a broken Robot.

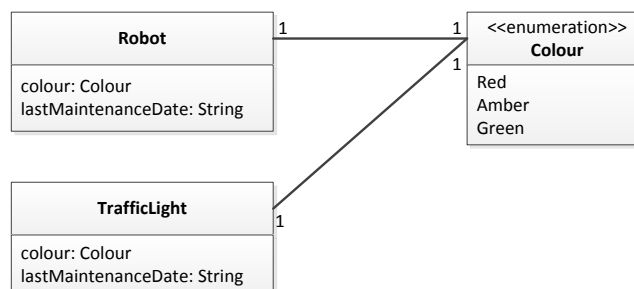


Figure 6.7: Robot and TrafficLight have the same attributes.

editors as discussed in Section 5.7.3 (p. 78).

```

Individual: Red
  Types: Colour
Individual: Amber
  Types: Colour
Individual: Green
  Types: Colour
DifferentIndividuals: Red, Amber, Green
Class: Colour
  EquivalentTo: {Red, Amber, Green}
Class: ColourDomain
  EquivalentTo: Robot or TrafficLight
Class: LastMaintenanceDateDomain
  EquivalentTo: Robot or TrafficLight
ObjectProperty: colour
  Domain: ColourDomain
  Range: Colour
DataProperty: lastMaintenanceDate
  Domain: LastMaintenanceDateDomain
  Range: String
Class: Robot
  SubClassOf: colour exactly 1 Thing
  SubClassOf: lastMaintenanceDate exactly 1 String
Class: TrafficLight
  SubClassOf: colour exactly 1 Thing
  SubClassOf: lastMaintenanceDate exactly 1 String

```

(6.5)

In accordance with step 2 of the formal scenario testing approach on p. 93, the reasoner is run on listing (6.5) to ensure that it is consistent. As shown in Figure 6.8, running the reasoner on listing (6.5) does not result in the inference that the classes `Robot` and `TrafficLight` are equivalent. The reasoner does however infer that the class `Robot` is a subclass of the class `ColourDomain`, which is equivalent to the anonymous class `Robot or TrafficLight`. The inference that `Robot` is a subclass of `Robot or TrafficLight` is not as meaningful as one would hope it to be. Indeed, for any classes `X` and `Y` it holds that class `X` is a subclass of `X or Y`. In particular, this inference does not enforce that individuals of `X` are necessarily individuals of `Y`.

If reasoning is done over the ontology represented by the TBox of listing (6.5) and the ABox of listing (6.6), the inferences as shown in Figures 6.9 (p. 101) and 6.10 (p. 101) are obtained. Again, these inferences fall short of showing that the classes `Robot` and `TrafficLight` are equivalent. However, the inferences obtained from Figures 6.9 and 6.10 make explicit that an

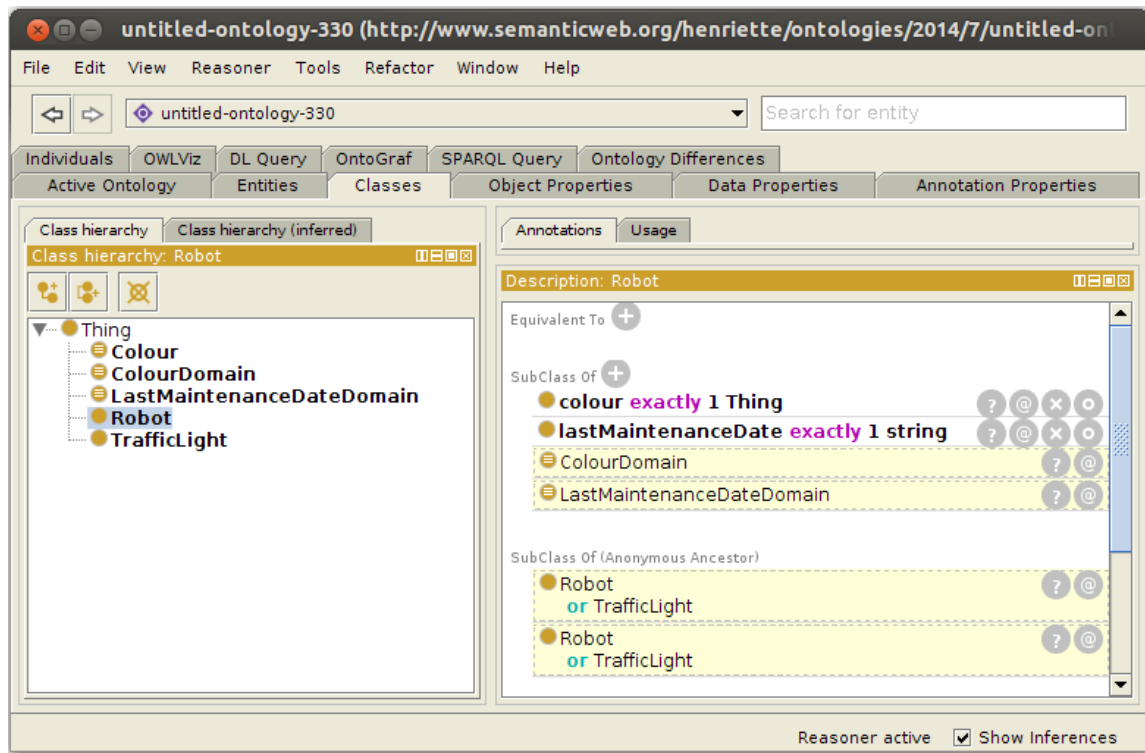


Figure 6.8: The classes Robot and TrafficLight are not equivalent.

individual that has the properties `colour` and `lastMaintenanceDate` can be of either type Robot or TrafficLight. Consequently, an individual that has the properties `colour` and `lastMaintenanceDate` is considered to be of both the types `ColourDomain` and `LastMaintenanceDateDomain` (see Figure 6.9). Since both the classes `ColourDomain` and `LastMaintenanceDateDomain` are equivalent to the union of the classes `Robot` and `TrafficLight` (see Figure 6.10), an individual that has the properties `colour` and `lastMaintenanceDate` can either be a Robot or a TrafficLight.

```

Individual: product
  Facts:
    colour Amber,
    lastMaintenanceDate "2014-08-07" string
  
```

(6.6)

The inference that individuals with the properties `colour` and `lastMaintenanceDate` can be considered to be either robots or trafficlighs indicates that there is a redundancy present in the ontology that represents the associated UML class diagram. By extension, this implies that either the `Robot` or `TrafficLight` class is redundant in the associated UML class diagram of Figure 6.7 (p. 98). This redundancy represents an unexpected result. According to step 3 of the formal scenario testing approach on p. 93, this requires the redesign of the

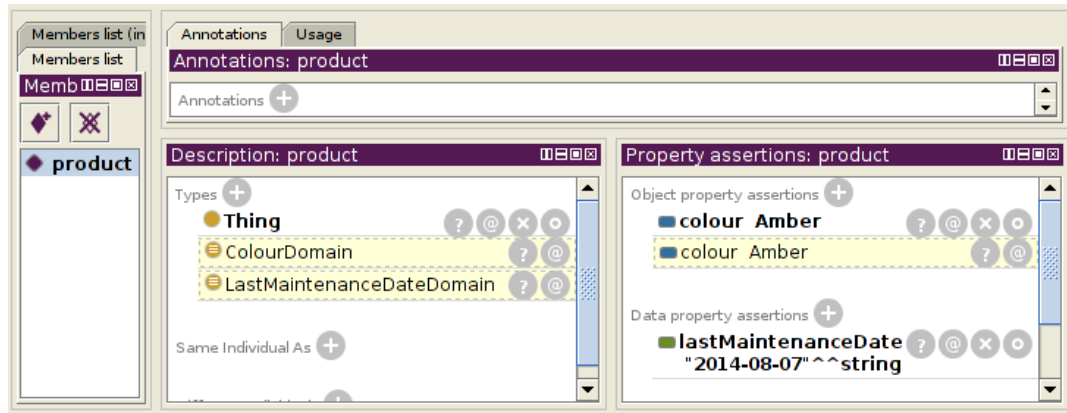


Figure 6.9: An individual with properties `colour` and `lastMaintenanceDate`.

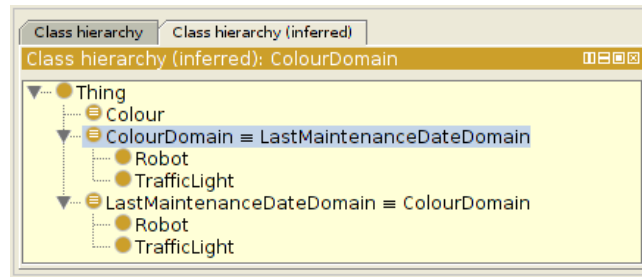


Figure 6.10: The classes `ColourDomain` and `LastMaintenanceDateDomain` are equivalent.

UML class diagram of Figure 6.7 (p. 98).

For the redesign it is sufficient to simply remove the `TrafficLight` class (or alternatively the `Robot` class) from the UML class diagram in Figure 6.7. The associated OWL 2 translation for the UML class diagram consisting only of the `Robot` and `Colour` classes is given in listing (6.7). Applying the scenario in listing (6.6) on the TBox in listing (6.7) results in the inference shown in Figure 6.11. The inference that an individual with the properties `colour` and `lastMaintenanceDate` is of type `Robot` is an expected result. This completes the formal scenario testing process.

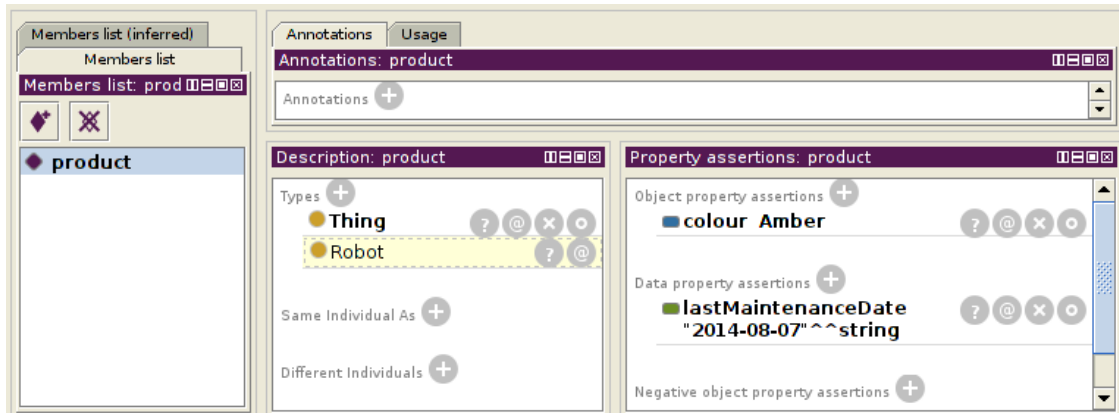


Figure 6.11: An individual is inferred to be of type Robot.

```

Individual: Red
  Types: Colour
Individual: Amber
  Types: Colour
Individual: Green
  Types: Colour
DifferentIndividuals: Red, Amber, Green
Class: Colour
  EquivalentTo: {Red, Amber, Green}
ObjectProperty: colour
  Domain: Robot
  Range: Colour
DataProperty: lastMaintenanceDate
  Domain: Robot
  Range: String
Class: Robot
  SubClassOf: colour exactly 1 Thing
  SubClassOf: lastMaintenanceDate exactly 1 String

```

(6.7)

6.2.4 Repairs for Formal Scenario Testing

Justification and abduction were discussed in Section 3.3.2 (p. 47). Recall that the purpose of justifications is to explain a given entailment for a given ontology. Abduction, on the other hand, will try to guess what axioms are missing from an ontology to ensure that a particular entailment follows from the ontology. This section discusses how justification and abduction can be used in finding repairs when using formal scenario testing. Four situations need to be considered.

Firstly, according to the formal scenario testing approach (see Section 6.1 p. 92), before

scenario testing can commence, the TBox of the DL translation of the UML class diagram must be consistent. If the TBox is inconsistent, justifications can be used to explain the inconsistency and based on the justifications, a repair can be constructed as explained in Section 3.3.2 (p. 47).

The second situation is when a consistent or classification scenario test gives an unexpected result. Since classification can be reduced to ontology consistency (see Section 3.3.1 p. 46), classification scenario tests are reducible to consistent scenario tests. When a consistent scenario test is inconsistent, the source of the inconsistency will be the ABox, since the assumption is that the TBox is consistent (see Section 6.1 p. 92). In typical justification and repair reasoning tasks, a first step is to remove the ABox [66]. For formal scenario testing this is not an option since the ABox represents a scenario test, which represents an exemplar of the business requirements. Rather, justifications can be determined, but instead of removing assertions from the ABox as a repair, axioms have to be removed from the TBox.

Thirdly, for an inconsistent scenario test, the expectation is that the ABox must be the cause for the ontology to be inconsistent, since again, it is the assumption that the TBox is consistent (see Section 6.1 p. 92). If the ontology turns out to be consistent for an inconsistent scenario test, it is because the ABox is consistent rather than inconsistent. In the case of inconsistent scenario tests, it is often the case that the TBox is not restrictive enough. In particular, the TBox permits models that are not permissible in a given business context. Due to the monotonicity of DLs, in order to reduce the number of models that are permitted by the TBox, axioms have to be added to the TBox (see Section 3.1.6 on p. 39). Hence, this can be stated as the following abductive reasoning problem: what axioms need to be added to the TBox such that the ABox will be inconsistent?

Lastly, throughout this discussion there has been a reluctance to make changes to the ABox. It is certainly possible that the ABox may have been constructed incorrectly. However, from a formal scenario testing perspective, an error in the ABox represents a fundamental lack of understanding of the business requirements. If it is suspected that this is indeed the case, it is best to consult with the domain expert and confirm the business requirements before continuing. Hence, automated reasoning will not be of help in this situation.

6.3 Guidelines

UML class diagrams are based on the CWA and the UNA (see Section 3.1.6 on p. 39), while DLs and OWL 2 adopt the OWA and not the UNA. Sections 6.3.1 and 6.3.2 (p. 106) give some guidelines for dealing with OWA and UNA respectively. Section 6.3.3 (p. 106) gives guidelines regarding how to structure scenario tests in an ontology editor such as for instance Protégé.

6.3.1 Dealing with OWA

In this section some guidelines are provided for dealing with the OWA (see Section 3.1.6 on p. 39) of DLs and OWL 2 in representing UML class- and object diagrams, which adopt the CWA. To bridge the OWA and CWA gap, scenario tests used in formal scenario testing have to make known information explicit. In this section some examples are given to illustrate how this can be achieved in the context of formal scenario testing.

Asserting that an Individual is not of a given Type

Section 6.2.3 (p. 97) illustrated how formal scenario testing can be utilized to detect that the properties `colour` and `lastMaintenanceDate` both belong to the classes `Robot` and `TrafficLight`. A slight variation is to assume the situation where a modeller already knows that a class `Robot` exists, which has the properties `colour` and `lastMaintenanceDate`. The question that the modeller will like to be answered is whether another class (which is not a robot) exists, which have the same properties.

This question can be answered using the scenario of listing (6.8). Since the modeller is looking for a class that is not a robot, the type of the individual `isThereSomethingElse` is explicitly stated as `not (Robot)`.

Applying this scenario to the TBox of listing (6.5) on p. 99 (which contains both a `Robot` and a `TrafficLight` class) results in the reasoner inferring that the individual `isThereSomethingElse` must be of type `TrafficLight`. This is shown in Figure 6.12. However, when scenario (6.8) is also applied to the TBox of listing (6.7) on p. 102, which contains only a `Robot` class, it results in the inconsistency of which the explanation is given in Figure 6.13. Moreover, the inference of Figure 6.12 states that there is a class called `TrafficLight` that is not a `Robot`, which has the properties `colour` and `lastMaintenanceDate`, while Figure 6.13 infers that such a class does not exist, hence the inconsistency.

```

Individual: isThereSomethingElse
Types: Thing,
      not (Robot)
Facts: colour Amber,
      lastMaintenanceDate "2014-08-11" string

```

(6.8)

Asserting that an Individual does not have a given Property

The UML class diagram of Figure 6.5 (p. 97) states that instances of the `Robot` class are always expected to have exactly one `colour` attribute. Scenario (6.4) on p. 97 illustrates that the reasoner can detect when more than one `colour` is assigned to an individual of type `Robot`.

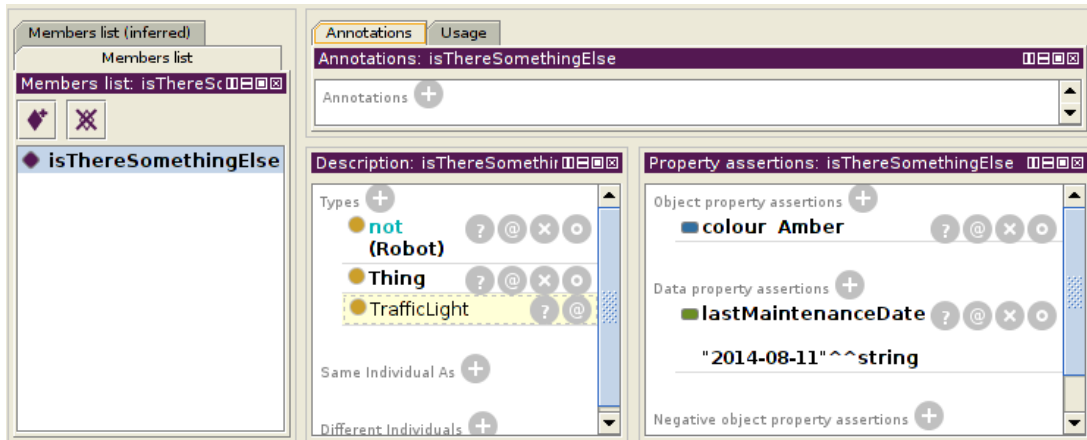


Figure 6.12: An individual is inferred to be of type TrafficLight.

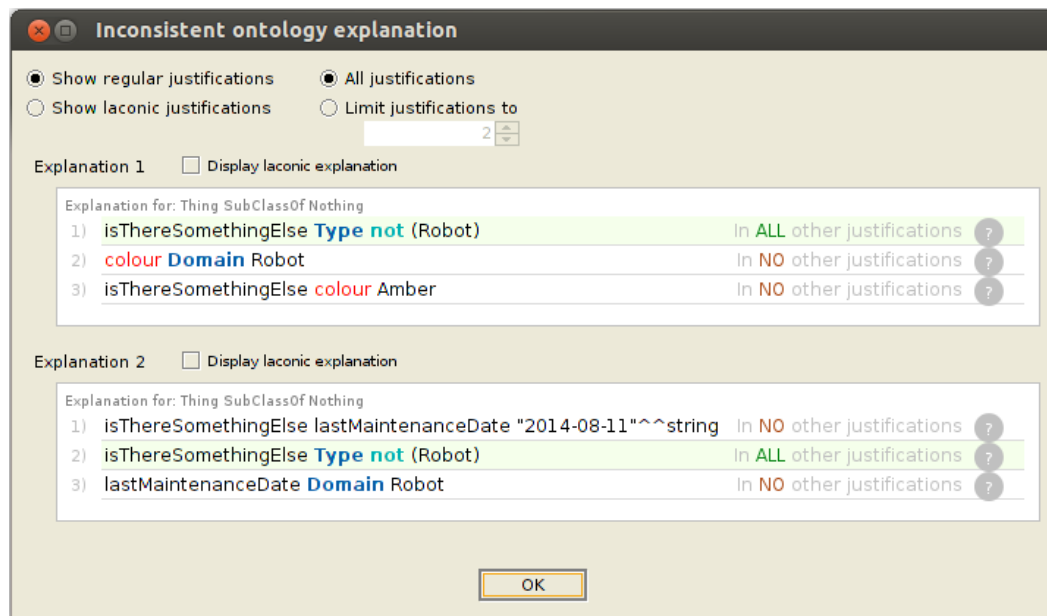


Figure 6.13: Explanation for the inconsistent scenario test in (6.8).

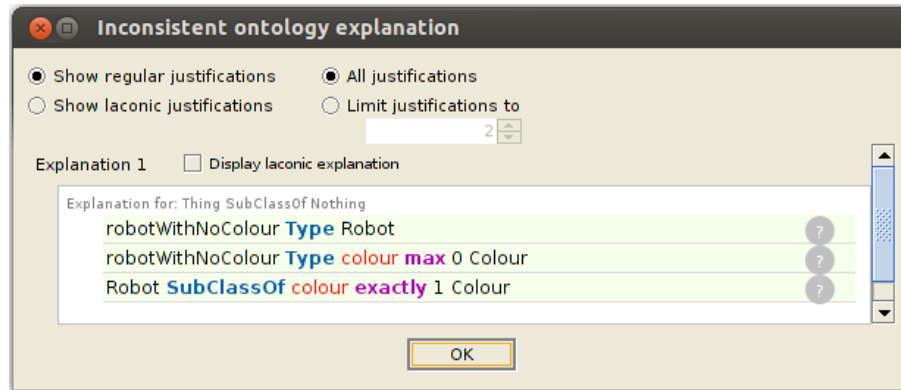


Figure 6.14: Explanation for a robot with no colour in (6.9).

As such scenario (6.4) on p. 97, which assigns the colours red, amber and green to a robot, results in the inconsistency of which the explanation is given in Figure 6.6 (p. 98).

What was not shown is that not assigning a `colour` property to an individual of type `Robot` will also result in an inconsistency. This can be achieved by applying scenario (6.9) to the TBox of listing (6.3) on p. 97. To state that the individual `robotWithNoColour` is a robot that does not have a colour, `robotWithNoColour` is defined as being of both type `Robot` and `colour max 0 Colour`. This scenario results in the expected inconsistency of which the explanation is given in Figure 6.14.

```

Individual: robotWithNoColour
Types: Robot,
       colour max 0 Colour
    
```

(6.9)

6.3.2 Dealing with UNA

Listing (6.3) on p. 97 defined the TBox for the UML class diagram of Figure 6.5 (p. 97). This listing includes the assertion `DifferentIndividuals: Red, Amber, Green`. In the absence of this assertion, scenario (6.4) on p. 97 (which assigns the colours red, amber and green to a robot) will not be inconsistent. This is because DLs and OWL 2 do not adopt the UNA (see Section 3.1.6 on p. 39). Hence it is possible that the individuals `Red`, `Amber` and `Green` represent the same individual.

6.3.3 Structuring Scenario Tests in Protégé

Throughout this dissertation formal scenario testing is applied using Protégé [1]. In using Protégé for formal scenario testing, it is often beneficial to structure Protégé files in a certain way. Experiences in this regards are shared here. In the discussion to follow it is important to note that when various formal scenario testing techniques are applied, the aim is not only to

confirm consistency, inconsistency or classification of the ontology, but also to ensure that the ontology is consistent, inconsistent or classified in a particular way for the correct reasons. In order to realize this requirement, the files of a Protégé project needs to be structured in a particular way.

Firstly, since anything can be entailed from an inconsistent ontology, no meaningful conclusions can be drawn from it [56]. It therefore makes sense to keep each inconsistent scenario in a separate file and reason on it separately. By keeping inconsistent scenarios in different files and reasoning on them separately, it can be ensured that the explanations for a particular inconsistent scenario pertains only to the inconsistent scenario that is currently considered. This makes it easier for a modeller to check that the scenario is inconsistent for the appropriate reasons.

Secondly, for the same reason it does not make sense to have different inconsistent scenarios in the same file, it does not make sense to mix inconsistent scenarios with either consistent or classification scenarios.

Thirdly, it is often more efficient from a modeller's perspective to have numerous consistent and classification scenarios in the same file. For a modeller this can be more efficient because rather than running the reasoner across multiple scenario one-by-one, it can be run once across a number of scenarios. However, different closely related consistent and classification scenarios can influence each other such that they may obscure the explanations as to why a particular scenario is consistent or classified in a specific way. The guideline in this regard is too keep consistent and classification scenarios together as long as it is relatively easy to make sense of the explanations provided. The moment it becomes difficult to understand explanations, it may make sense to split the scenarios across different files.

The last consideration in structuring ontologies for formal scenario testing follows from the need to spread scenario tests across different files. The different files representing scenario tests correspond to different ABoxes in DL parlance (or object diagrams in UML parlance). All these ABoxes (respectively object diagrams) have the same TBox (respectively class diagram). Therefore the TBox (respectively class diagram) is defined in a separate file, which is imported into each of the different files representing the different scenario tests.

In Figure 6.15 a graphical representation is provided of how to structure files in Protégé. Each rectangle represent a different file in Protégé. The rectangle marked **TBox: Class diagram** represents the TBox obtained when the class diagram is translated to OWL 2. The rectangles marked **ABox 1**, **ABox 2**, ..., **ABox n**, **ABox n+1**, ..., **ABox n+m** represent different ABoxes, each corresponding to a single scenario test or group of scenario tests. Each file representing an ABox imports the file representing the TBox, thereby forming the ontologies **Ontology 1**, **Ontology 2**, ..., **Ontology n**, **Ontology n+1**, ..., **Ontology n+m** respectively.

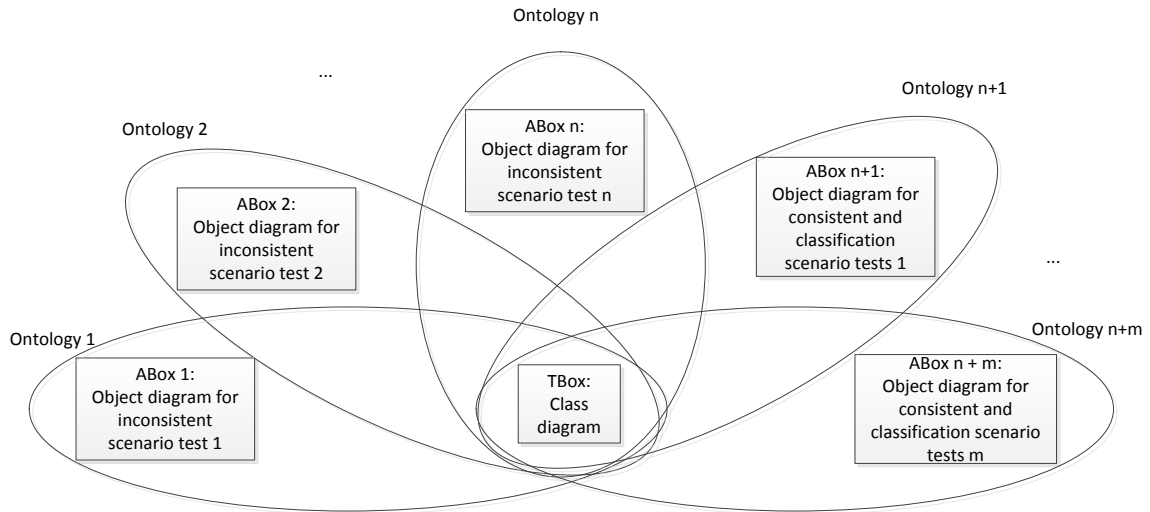


Figure 6.15: The same TBox is reused for different ABoxes.

6.4 A Case Study

This section starts by explaining the business requirements for a portion of the business from a real-world project in the hospitality industry in Section 6.4.1. Section 6.4.2 (p. 109) proceeds by providing the UML class diagram as it was modelled initially. It explains why this initial UML class diagram is problematic and it provides a UML class diagram which addresses the deficiencies of the naïve UML class diagram. Section 6.4.3 (p. 111) shows how formal scenario testing can be employed to validate that the improved UML class diagram of Section 6.4.2 (p. 109) indeed does address the deficiencies mentioned. In Section 6.4.4 (p. 114) preliminary feedback is given on experiences in using formal scenario testing.

6.4.1 Business Domain

The example business domain that is described here, is a simplified version of the real-world software project that was done for a South African hotel group. On this project formal scenario testing was employed to detect and rectify conceptual modelling errors during the requirements engineering phase of the SDLC.

In the hospitality industry the business requirements around calculating the rate that needs to be charged when a reservation is made, generally referred to as a room rate, can be extremely complex. Before a room rate can be calculated, the different configurations that are used to determine these rates have to be specified. In this example, the focus is on the UML class diagram representing rate configurations. In the following a brief description is given of the relevant concepts and business constraints.

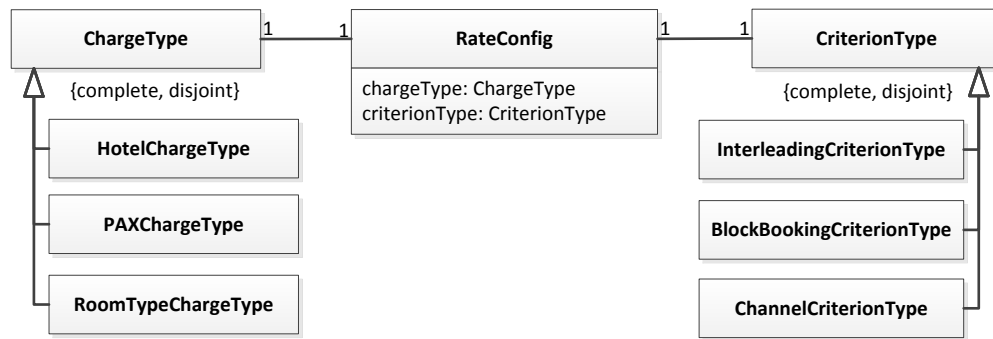


Figure 6.16: The initial UML class diagram for rate configuration.

The basic rate that is applicable to a room can be determined in three different ways. Firstly, a flat rate can be charged across all rooms in the hotel. Secondly, the rate can be charged based on the number of guests who form part of the booking. In the hospitality industry this is often referred to as PAX. Lastly, the rate can be determined based on the kind of room that is booked, i.e. single room or suite.

The basic rate that applies to a room can be adjusted based on additional criteria that apply to a room. For instance, a different rate will apply when two rooms are booked that are connected by an interleading door. When a reservation is made through a partner network (commonly referred to as a channel) a discounted rate may apply. Similarly when a guest makes a block booking, say of more than 10 rooms per night, a different rate may apply. In order to keep the example succinct, the assumption is made that basic rate charges are always adjusted.

Additional business rules are that PAX charges can only be adjusted by a channel criterion, while a room type charge can be adjusted by a block booking or a channel criterion. Hotel charges can be adjusted by any criteria. Failure to adhere to these business rules will give rise to opportunities for fraud.

6.4.2 Deficiencies of a Naïve UML Class Diagram and a Solution

Figure 6.16 presents the UML class diagram that was initially created to represent the rate configuration business requirements. The UML class diagram of Figure 6.16 is consistent when translated to OWL 2. Even so, this UML class diagram will allow combinations of charge types and criterion types that (based on the business requirements) should be disallowed. To make these deficiencies in the UML class diagram apparent, the possible scenarios are compiled in Table 6.1 and for each scenario it is indicated whether it is allowed or disallowed.

Figure 6.17 (p. 110) shows the UML class diagram redesigned to represent the business requirements concisely. The essence of the redesign is to make the combinations of charge types and criterion types that are allowed by the business explicit in the UML class diagram. This is achieved through a combination of class specialization and attribute redefinition, which

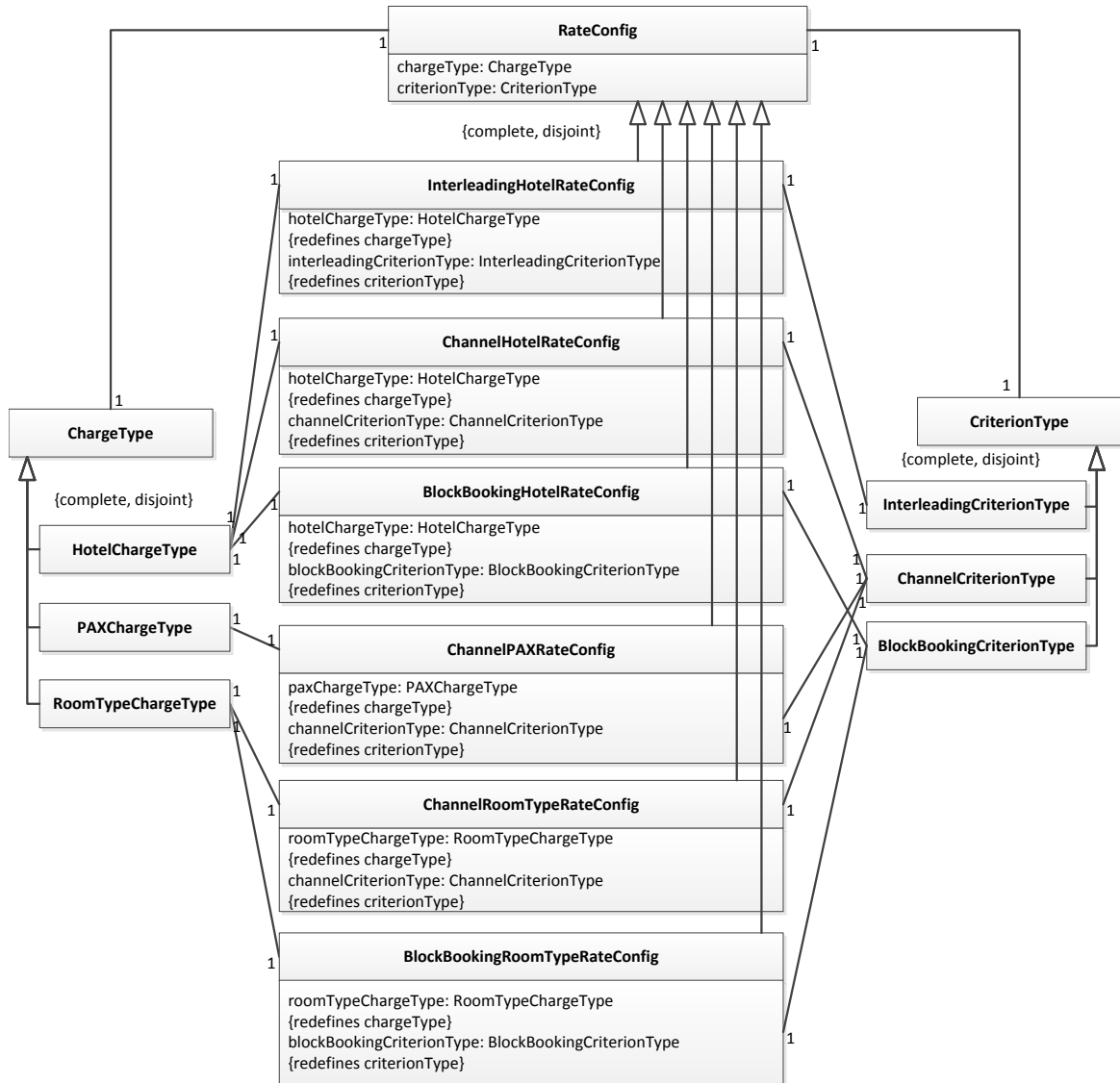


Figure 6.17: An accurate representation of the business requirements.

Table 6.1: Allowed/disallowed rate configuration scenarios

No	Criteria Type			Charge Type			Allowed or Disallowed
	Interleading	Channel	Block booking	Hotel	PAX	Room type	Scenario
1	X			X			Allowed
2		X		X			Allowed
3			X	X			Allowed
4	X				X		Disallowed
5		X			X		Allowed
6			X		X		Disallowed
7	X					X	Disallowed
8		X				X	Allowed
9			X			X	Allowed

was discussed respectively in Sections 2.1.4 (p. 17) and 2.1.5 (p. 18). The next section shows how this UML class diagram can be validated using formal scenario testing.

6.4.3 Validating the UML Class Diagram

Protégé is used as was discussed in Section 6.3.3 (p. 106) to validate that scenarios that ought to be inconsistent are indeed inconsistent. The TBox is defined based on the class diagram in Figure 6.17 for which the full listing is provided in Appendix A (p. 154). The translation of the UML class diagram is based on the translations provided in Chapters 4 and 5. Note that the translation of attributes with the same name appearing in different classes are translated as a single property of which the domain consists of the union of the classes the attribute appears in (as is described in Section 5.7.3 on p. 78).

For illustrating how formal scenario testing can be used to validate that a disallowed scenario is indeed inconsistent, the disallowed scenario is considered where a `PAXChargeType` is combined with an `InterleadingCriterionType` as shown in (6.10). The individuals `pax` and `interleading` are defined to be of type `PAXChargeType` and `InterleadingCriterionType` respectively. For the disallowed scenario the individual `interleadingPAXRateConfig`, which is of type `RateConfig` is introduced and the disallowed combination of `paxChargeType` and `interleadingCriterionType` properties are assigned using the individuals `pax` and `interleading` respectively.

Running the reasoner on this disallowed scenario results in the expected inconsistent ontology. The explanations for the inconsistency is shown in Figure 6.18. These explanations essentially state that based on the properties `paxChargeType` and `interleadingCriterionType` the type of the individual `interleadingPAXRateConfig` is inferred to be of type

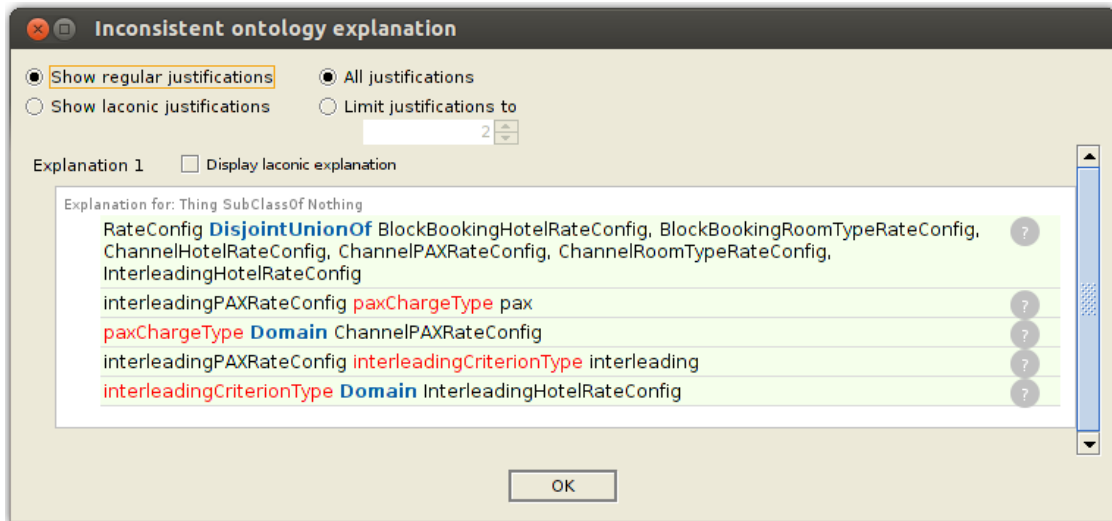


Figure 6.18: Explanation for the disallowed scenario.

ChannelPAXRateConfig and InterleadingHotelRateConfig, but since the classes ChannelPAXRateConfig and InterleadingHotelRateConfig are disjointed, there is an inconsistency.

```

Individual: pax
  Types: PAXChargeType
Individual: interleading
  Types: InterleadingCriterionType
Individual: interleadingPAXRateConfig
  Types: RateConfig
Facts:
  paxChargeType pax
  interleadingCriterionType interleading
  
```

(6.10)

As stated in Section 6.2.1 (p. 94) formal scenario testing expects allowed scenarios to be consistent. It should for example be possible to configure a RateConfig that combines an InterleadingCriterionType and a HotelChargeType. The listing for this scenario is provided in (6.11). Running the reasoner on this scenario confirms that this scenario is indeed consistent as shown in Figure 6.19.

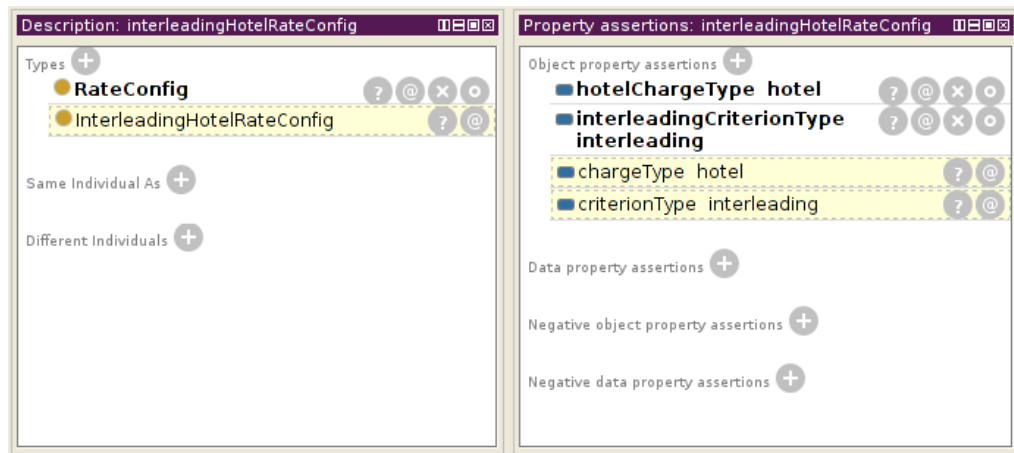


Figure 6.19: `interleadingHotelRateConfig` is of type `InterleadingHotelRateConfig`.

```

Individual: hotel
  Types: HotelChargeType
Individual: interleading
  Types: InterleadingCriterionType
Individual: interleadingHotelRateConfig
  Types: RateConfig
Facts:
  hotelChargeType hotel
  interleadingCriterionType interleading
  
```

(6.11)

```

Individual: hotel
  Types: HotelChargeType
Individual: interleading
  Types: InterleadingCriterionType
Individual: interleadingHotel
Facts:
  hotelChargeType hotel
  interleadingCriterionType interleading
  
```

(6.12)

A classification scenario test can be created by stating specific properties that an individual has without specifying the type of the individual. Hence, the reasoner is given the opportunity to infer the type of the individual purely based on the properties assigned to the individual. An example of this is illustrated in listing (6.12). Note that the difference between listing (6.11) and (6.12) is that the type of individual `interleadingHotelRateConfig` is set while for individual `interleadingHotel` it is not set. Running the reasoner on listing (6.12) allows detection of classes outside of the `RateConfig` class hierarchy that also have attributes of

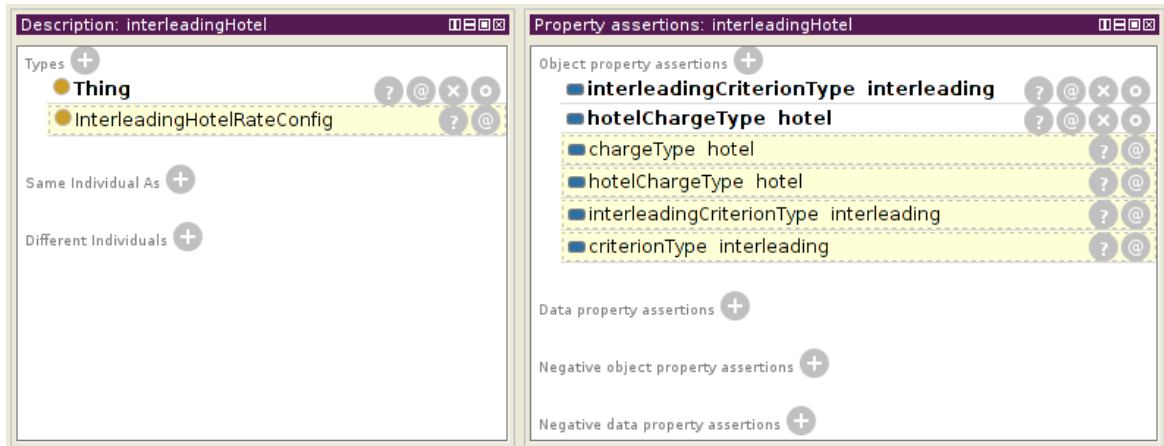


Figure 6.20: `interleadingHotel` is of type `InterleadingHotelRateConfig`.

type `HotelChargeType` and `InterleadingCriterionType`.

Figure 6.20 shows that individual `interleadingHotel` is inferred to be only of type `InterleadingHotelRateConfig`. This proves that there is only one class in the UML class diagram that have attributes of type `HotelChargeType` and `InterleadingCriterionType`. As such classification scenario tests can be used to detect potential redundancies.

6.4.4 Adoption and Preliminary Feedback

As explained at the beginning of this case study, the example discussed here forms part of a larger real-world software project for a hotel group in South Africa. On this project formal scenario testing was used to help model the calculation of room rates. Here feedback is given on experiences gained in using formal scenario testing on this project.

The business analyst (BA) on this project was responsible for creating the UML class diagrams. This BA had extensive experience in UML and in specific UML class diagrams, but no prior knowledge regarding OWL 2 or DLs. This BA was given instruction in translating UML class diagrams to Protégé.

The BA and client started with an initial UML class diagram that was translated to OWL 2. Using Protégé they set up a number of scenario tests, which quickly showed that the initial UML class diagram did not represent the required business rules adequately. Interestingly, at this point they abandoned the UML class diagram in favour of Protégé. Occasionally they drew portions of the UML class diagram to help guide their thinking. In this manner they incrementally created the UML class diagram by validating each increment with a number of scenario tests. Only when the scenario tests confirmed that the UML class diagram corresponds with the business requirements, did they extend the conceptual schema to include additional concepts from the business domain. Once they completed the UML class diagram in this fashion in Protégé, the BA translated the classes and relationships to a UML class

diagram.

At the completion of this process the resulting UML class diagram consisted of 62 classes, which have been validated by 100 scenario tests. The scenario tests consisted of 43 allowed, 47 disallowed and 10 classification scenario tests. The BA used the resulting UML class diagram to explain the business logic to the development team. Both developers and testers frequently referred to this UML class diagram throughout the development process.

Feedback from this project on using formal scenario testing highlights some opportunities for improvement.

1. There is a need for tools to assist with the translation between UML and OWL 2. Zedlitz, et. al. [118] have suggested such a tool, but it will need to be extended to fit the needs of scenario testing.
2. Formal scenario testing was instrumental in providing a clear understanding of the business requirements on this project, which benefited both developers and testers. However, the UML class diagram relied on multiple inheritance, which cannot be represented directly in many programming languages [104] and it is not clear how to translate notions like `{disjoint, complete}` into code. Guidelines in this regard will be valuable.

About a year after the initial rates calculation module was implemented, there was an additional need to add business rules and functions. Changing the UML class diagram for the rates calculation module resulted in a minimum of rework. The main reason for this seems to be the fact that the new business capabilities could be added without changing any existing classes. Thus, new capabilities have been added by either adding completely new classes or extending existing classes. It seems that the key factor that enabled this efficiency is that the classes of the rates calculation module have model class cohesion. Accordingly, each class represents a single cohesive concept, as was discussed in Section 2.2 (p. 21). Indeed, high cohesion has been positively correlated with high maintainability of software systems [86]. It is therefore conceivable that high cohesion of the rates calculation UML class diagram was the key factor in ensuring its maintainability.

6.5 Contribution and Related Research

In this section a synopsis is given of the research contributions discussed in this chapter. These contributions are contextualized by a review and evaluation of related research.

6.5.1 Contribution

Formal scenario testing provides a means for the early detection of errors of business intent. An error of business intent is where there is a mismatch between the semantics of the UML

class diagram and the desired business requirements. The hope is that through enabling early detection of errors of business intent, the identified errors can be remedied before the actual construction of the software system commences. Formal scenario testing can also be employed to validate that errors of business intent have not been committed. Consequently, formal scenario testing can be used to confirm that the UML class diagram agrees with the business requirements. Detection and validation are enabled through the presence of a domain expert who provides scenario tests that serve as exemplars of either scenarios that are allowed or disallowed based for the business requirements.

In formal scenario testing detection and validation are based on the reasoning capabilities of DLs and OWL 2 (see Chapter 3). Chapter 4 explained how UML class diagrams can be translated to DLs (respectively OWL 2). In particular, the classes in UML class diagrams are translated to concepts in DLs (respectively classes in OWL 2) and associations/attributes in UML class diagrams are translated to roles in DLs (respectively properties in OWL 2). Formal scenario testing extends this work by allowing reasoning on scenario tests, which can be expressed as UML object diagrams. That is, instances in UML object diagrams are translated to individuals in DLs (respectively individuals in OWL 2).

Below a synopsis is given of the contributions of this chapter.

1. A step-wise process is defined, which explains how formal scenario testing can be applied (Section 6.1 p. 92).
2. Techniques for defining scenario tests for formal scenario testing are described. These techniques are based on the consistency checking and classification reasoning procedures of DLs (Section 6.2 on p. 94).
3. An evaluation is done of the appropriateness of justification and abduction reasoning procedures in the context of formal scenario testing (Section 6.2.4 on p. 102).
4. UML class diagrams adopt the UNA and the CWA while DLs and OWL 2 adopt the OWA and not the UNA. Guidance is given in how to deal with this mismatch when using formal scenario testing (Sections 6.3.1 (p. 104) and 6.3.2 (p. 106)).
5. Practical guidance is given for how to structure ontologies in an ontology editor like Protégé (Section 6.3.3 on p. 106).
6. A small case study is described where formal scenario testing was used on a real-world project. This case study indicates that formal scenario testing can be helpful in creating a conceptual schema that has model cohesion, which is beneficial for the long term maintainability of the conceptual schema (Section 6.4 p. 108).

6.5.2 Related Research

Formal scenario testing extends the research of Cali, et. al., Berardi, et. al. [25, 13] and Zedlitz, et. al. [118], which translated UML class diagrams to TBoxes for DLs and OWL 2. In formal scenario testing UML object diagrams (which represents scenario tests) are translated to ABoxes in DLs and OWL 2 respectively. The main motivation for translating UML class diagrams to DLs/OWL 2 is to verify that the UML class diagram is consistent and free of redundancies [13]. The main motivation for translating UML object diagrams to DLs/OWL 2 is to validate that the business intent of the UML class diagram agrees with the business requirements.

Various approaches exist for validating UML class diagrams based on generated instances [20, 23, 110]. Cabot, et. al. [23] encodes UML class diagrams as constraint satisfaction problems (CSP) and then generates instances of the model using their UMLtoCSP tool, which passes the instances to a constraint solver for validation. UMLtoCSP generates a UML object diagram of the object instances that satisfies the UML class diagram. Soeken, et. al. [110] follow a similar approach using C++ code to generate instances and a SAT solver to do validation. For both approaches decidability is achieved by definition of a finite solution space and therefore both approaches are decidable, but incomplete. Hence, results are only conclusive when a solution is found. When a solution is not found, a solution may still exist in some other finite solution space [23, 110].

Braga, et. al. [20] apply scenario testing to OntoUML conceptual models, which are translated into Alloy, a logic based language. OntoUML is a UML profile that extends the UML class diagram metamodel with Unified Foundation Ontology (UFO) elements. UFO defines the ontological foundations for the most fundamental concepts in structural conceptual modeling [20]. Alloy is defined as “a structural modeling language based on first-order logic, for expressing complex structural constraints and behavior” [65]. In the approach of Braga, et. al. the Alloy analyzer is used to automatically generate instances and counterexamples of the model which are presented to the modeller [20]. The Alloy logic is based on first-order logic (and in particular relational calculus), which gives rise to undecidability. Tractability is achieved by specifying a scope, which means a counterexample may possibly be found given a larger scope [65].

Formal scenario testing is different in the following ways. Firstly, since OWL 2 is based on the DL $\mathcal{SROIQ}^{(D)}$ extended with Easy Keys, reasoning on OWL 2 is decidable and complete [97, 58]. Thus, theoretically it is possible to get an answer on whether a knowledge base is consistent, but due to tractability concerns there is a practical limit to the size of the knowledge base on which reasoning is feasible [47]. Secondly, formal scenario testing relies on the presence of a domain expert for guiding the definition of scenario tests. None of the approaches mentioned cater for the explicit definition of scenario tests [20, 23, 110] and hence,

there is no way to ensure that scenario tests with high business impact are indeed considered. Furthermore, even when a model is consistent, it may not represent the business requirement accurately. The explicit specification of scenario tests can help alert the modeller to this occurrence.

Formal scenario testing shares some similarities with the research of Tort, et. al. [114]. Similar to formal scenario testing domain experts are employed to provide scenarios that are either expected to be consistent or inconsistent. For writing tests Tort, et. al. introduced the conceptual schema testing language (CSTL), which can run automated tests written for UML class diagrams with OCL constraints. Testing of a conceptual schema proceeds in a similar fashion as for unit testing of software. Thus, a test application is responsible for running a number of tests written in CSTL in sequence with the result of each test being compared to an expected result for which the test application reports on the result of each comparison.

Differences in the approach of Tort, et. al. [114] and formal scenario testing are explained next. In Tort, et. al. OCL constraints are explicitly included. In Section 5.9.2 (p. 81) it was explained that it may be possible to support OCL-Lite, but that this falls outside the scope of the current dissertation. Due to the inclusion of OCL constraints, Tort, et. al. are able to consider tests that validate the occurrence (or non-occurrence) of domain events. Automated testing, the ability to run sequences of tests, is also not currently possible with formal scenario testing. Formal scenario testing is likely to benefit from automated testing tool support that can generate a consolidated report on running the reasoner across all scenario tests (that is, consistent-, inconsistent- and classification scenario tests). CSTL, the programming language that Tort, et. al. use for writing tests, has no formal reasoning capabilities and hence cannot detect nor explain logical consistencies/inconsistencies and entailments beyond that which can be detected with tests. Since formal scenario testing is based on DLs, implicit consequences can be made explicit.

Chapter 7

Applying Formal Scenario Testing

In this chapter it is shown how formal scenario testing can be applied to improve the quality of UML class diagrams by detecting violations of the heuristics that were discussed in Section 2.2 (p. 21). Recall that for each heuristic in Section 2.2 (p. 21), an example UML class diagram was provided that illustrates how the heuristic is violated. This was followed by a redesigned UML class diagram, which corrects the violated heuristic. This chapter proceeds by first indicating how the heuristic violation can be detected and then shows how it can be confirmed that the correction is indeed free from the given heuristic violation.

Each of the following sections are dedicated to one modelling heuristic. To keep this chapter succinct and predictable, the same format is used for each of the sections. At the beginning of each section, the formal scenario testing technique(s) that are applied to detect and validate the heuristic, is stated. Each section is divided into “Detection” and “Validation” subsections. In the “Detection” subsection it is illustrated how the violation of the heuristic can be detected. In the “Validation” subsection it is shown how it can be validated that a class or classes do not suffer of the particular heuristic violation. At the beginning of each “Detection” and “Validation” subsection it is stated which figure refers in Section 2.2 (p. 21) and where the related OWL 2 translation can be found.

The detection and validation techniques that are defined using formal scenario testing represent the main contributions of this chapter. These are detection and validation techniques for:

1. separable class cohesion (see Section 7.1),
2. multifaceted class cohesion (see Section 7.2 p. 125),
3. non-delegated class cohesion (see Section 7.3 p. 127),
4. concealed class cohesion (see Section 7.4 p. 131),
5. low inheritance cohesion of attributes (see Section 7.5 p. 141) and
6. low inheritance cohesion of operations (see Section 7.6 p. 144).

7.1 Separable Class Cohesion

Formal Scenario Testing Technique: Classification

7.1.1 Detection

Figure: 2.11, p. 23

OWL 2 Translation: Appendix B.1, p. 157

Separable class cohesion is detected by creating an individual of type **Thing**. It is important not to assign a type other than **Thing** to the individual. Rather, specific properties are assigned to the individual that correspond to attributes and/or operations that are relevant for a particular concept. If a reasoner is allowed to classify this individual, it will determine the type of the individual based on the assigned properties. If this process is now repeated for a second individual representing a different concept than the first individual, the expectation is that the two individuals will be classified as different classes. This process is illustrated by the following example.

If an individual **pablo** of type **Thing**, with properties **employeeName**, **employeeCode** and **salary**, is added to the ontology in listing B.1 (p. 157), as shown in listing (7.1), it will cause the individual **pablo** to be classified to be of type **Employee** (see Figure 7.1). If another individual **projectX** of type **Thing** is added, to which project related properties are assigned, it is also classified to be of type **Employee**. The listing for the individual **projectX** is given in (7.2) and the related inference can be seen in Figure 7.2. The fact that the individuals **pablo** and **projectX**, which represent different concepts (respectively employee and project), are classified as belonging to the same class (**Employee**), is an indication that the class (**Employee** in this case) has separable class cohesion.

```
Individual: pablo
  Types: Thing
  Facts:
    employeeName "Pablo" string,
    employeeCode "1234" string,
    salary 100000
```

(7.1)

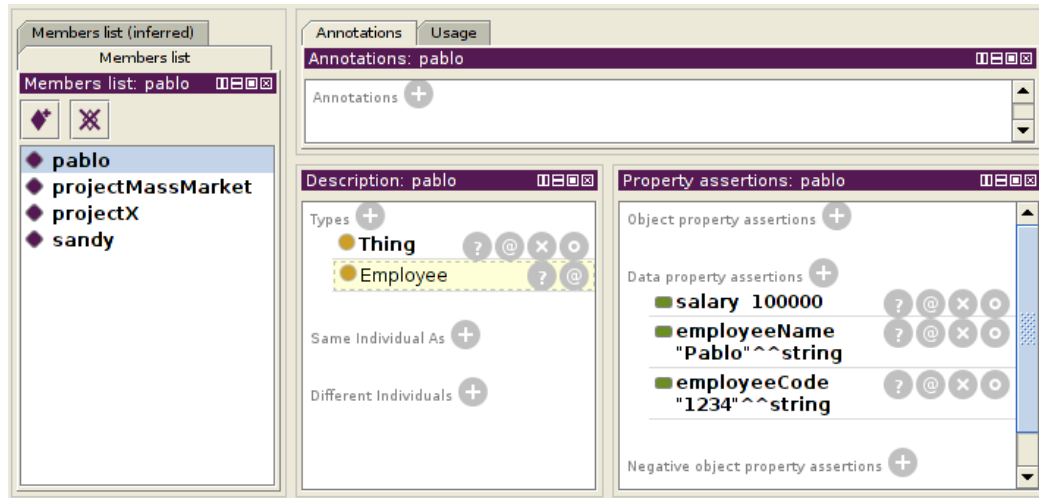


Figure 7.1: Classification of the individual pablo.

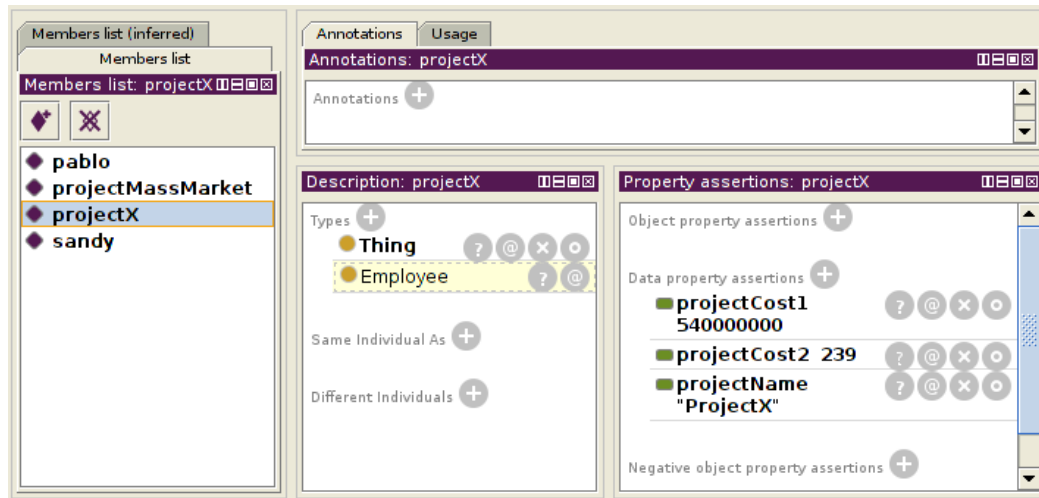


Figure 7.2: Classification of the individual projectX.

Individual: projectX

Types: Thing

Facts:

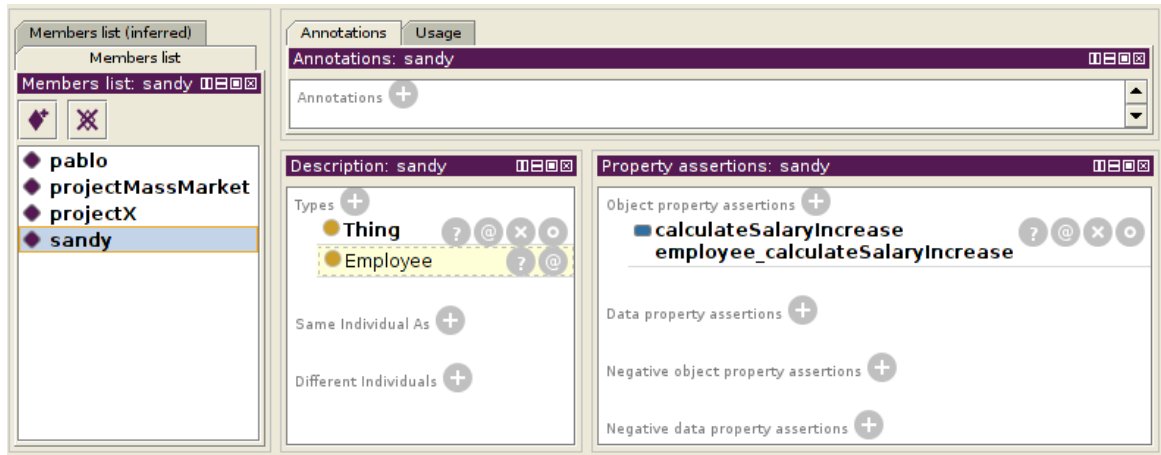
```

projectName "projectX" string,
projectCost1 540000000,
projectCost2 239

```

(7.2)

Separable class cohesion can also be detected by considering the operations that can be called on a class. An individual on which it is asserted that it is in a property relation with an individual of a class representing an operation, will be inferred to be of a class on which the operation can be called. This is illustrated with the following example.

Figure 7.3: Classification of the individual `sandy`.

Listing (7.3) defines an individual `employee_calculateSalaryIncrease` of type `Employee_calculateSalaryIncrease_integer`, which is the class that is introduced to represent the reified relation representing the operation `calculateSalaryIncrease(increase:Integer):Integer` (see Section 5.3 on p. 66). It states that the operation, represented by the class `Employee_calculateSalaryIncrease_integer`, can be performed by the individual `sandy` by asserting the fact that `sandy` is in the property relation `calculateSalaryIncrease` with the individual `employee_calculateSalaryIncrease`. Note that again the type `Thing` is assigned to the individual `sandy` to allow the reasoner to determine the class that `sandy` belongs to. The classification of `sandy`, based on the operation that can be performed on this individual, is shown in Figure 7.3.

In listing (7.4) it is stated that the operation `calculateProjectCost():Integer` can be performed on the individual `projectMassMarket`. Again classification is applied and from Figures 7.3 and 7.4 it is inferred that both the individuals `sandy` and `projectMassMarket` are of type `Employee`. Again this is an indication of the class `Employee` having separable class cohesion.

```

Individual: employee_calculateSalaryIncrease
  Types: Employee_calculateSalaryIncrease_integer
Individual: sandy
  Types: Thing
  Facts:
    calculateSalaryIncrease employee_calculateSalaryIncrease

```

(7.3)

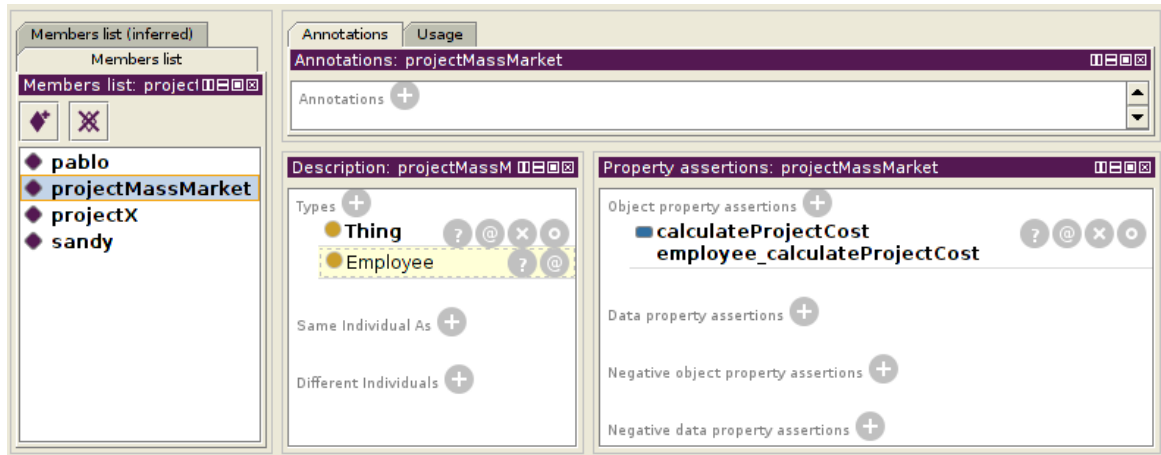


Figure 7.4: Classification of the individual projectMassMarket.

```

Individual: employee_calculateProjectCost
  Types: Employee_calculateProjectCost_integer
Individual: projectMassMarket
  Types: Thing
  Facts:
    calculateProjectCost employee_calculateProjectCost
  
```

(7.4)

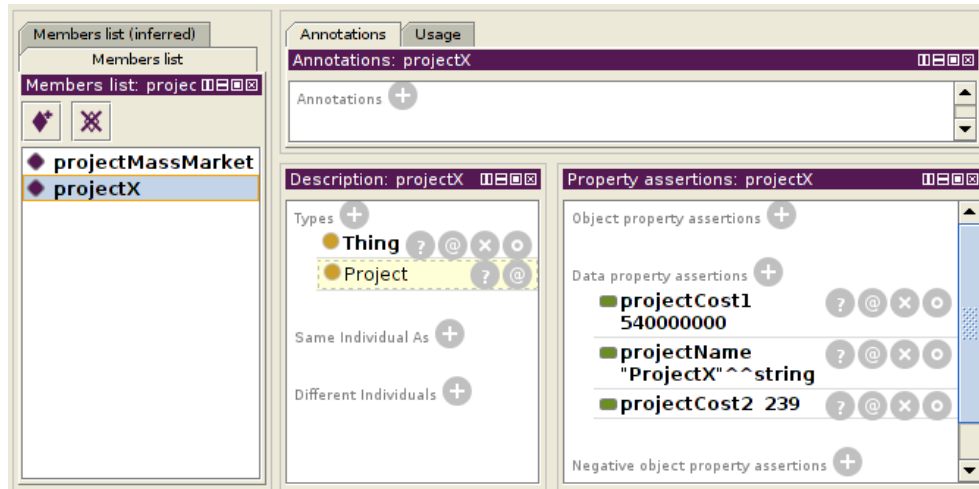
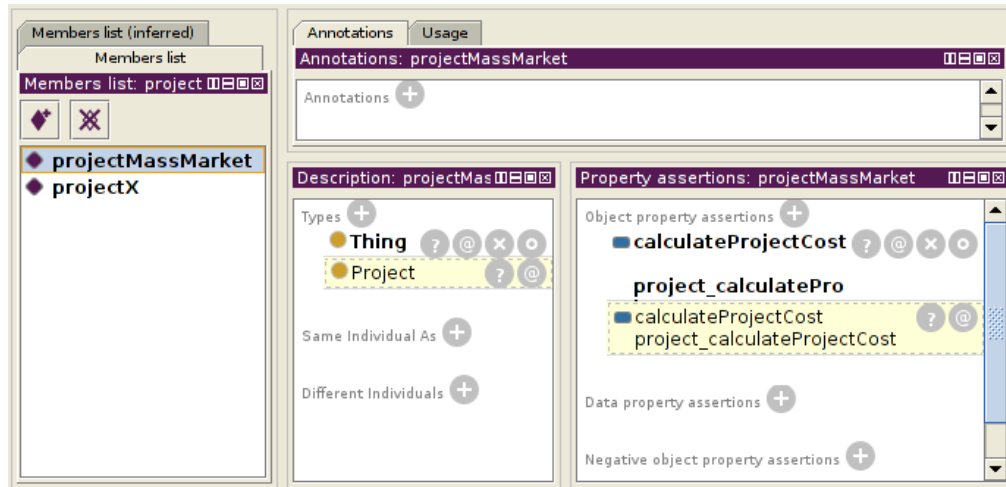
7.1.2 Validation

Figure: 2.13, p. 23

OWL 2 Translation: Appendix B.2, p. 159

When the scenarios of listings (7.1) on p. 120 and (7.2) on p. 121 are applied to the OWL 2 translation of the UML class diagram in which separable class cohesion has been corrected, the result of the inferences is most notable on the `projectX` individual. The individual `pablo` is still classified as being of type `Employee`, but the `projectX` individual is now classified as being of type `Project`. This is shown in Figure 7.5. Since the types of `pablo` and `projectX` are correctly inferred, it confirms that separable class cohesion has been removed from the associated UML class diagram.

When operations are considered, the scenario in listing (7.3) applies as-is on the redesigned conceptual schema. However, listing (7.4) have to be revisited. The revised scenario of listing (7.4) is given in listing (7.5).

Figure 7.5: The individual `projectX` is correctly classified.Figure 7.6: The individual `projectMassMarket` is correctly classified.

```

Individual: project_calculateProjectCost
  Types: Project_calculateProjectCost_integer
Individual: projectMassMarket
  Types: Thing
  Facts:
    calculateProjectCost project_calculateProjectCost

```

(7.5)

Applying the scenario in listing (7.5) on the OWL 2 translation of the redesigned UML class diagram now results in the individual `projectMassMarket` being classified as being of type `Project`, as illustrated in Figure 7.6.

7.2 Multifaceted Class Cohesion

Formal Scenario Testing Technique: Inconsistent

7.2.1 Detection

Figure: 2.15, p. 24

OWL 2 Translation: Appendix C.1, p. 161

Multifaceted class cohesion can be detected by applying a scenario similar to the scenario represented by the UML object diagram in Figure 2.16 (p. 24). The OWL 2 translation of this scenario is given in listing (7.6). In Section 2.2.2 (p. 22) it was stated that for this business context it is assumed that a company can only have one address. Hence, this is a scenario that is expected to be inconsistent since both Pablo and Sandy work at the same company (i.e. the CSIR), but the UML class diagram allows different addresses (i.e. `csirAddress` and `otherAddress`) to be assigned as the company address. When the reasoner is run on this scenario, it finds that it is consistent rather than inconsistent, which indicates that there is a modelling error in the UML class diagram.

Note that in scenario (7.6) it is necessary to explicitly state that the individuals `csirAddress` and `otherAddress` are different individuals. If this assumption is not made explicit, since DLs and OWL 2 do not adopt the UNA (see Section 6.3.2 on p. 106), the reasoner can infer that the individuals `csirAddress` and `otherAddress` represent exactly the same individual.

```
Individual: csirAddress
  Types: Address
  DifferentFrom: otherAddress
Individual: otherAddress
  Types: Address
  DifferentFrom: csirAddress

Individual: sandyContactInformation
  Types: ContactInformation
  Facts:
    companyName "CSIR",
    contactPerson "Sandy" string,
    companyAddress csirAddress,
    phoneNumber "1234" string
Individual: pabloContactInformation
  Types: ContactInformation
  Facts:
    companyName "CSIR",
    contactPerson "Pablo" string,
    companyAddress otherAddress,
    phoneNumber "5678" string
```

(7.6)

7.2.2 Validation

Figure: 2.17 p. 24

OWL 2 Translation: Appendix C.2, p. 162

The redesigned UML class diagram can be validated by setting up a scenario where different addresses are assigned to different individuals of type `Company`, but with all individuals having the same company name, e.g. “CSIR”. This is shown in listing (7.7). Running the reasoner on this scenario results in an inconsistency. This inconsistency confirms that the UML class diagram of Figure 2.17 (p. 24) is modelled tightly enough to disallow the scenario where the same company can have different addresses. The partial list of explanations of the inconsistency is shown Figure 7.7 (p. 128).

```

Individual: csirAddress
  Types: Address
  DifferentFrom: otherAddress
Individual: otherAddress
  Types: Address
  DifferentFrom: csirAddress

Individual: csir
  Types: Company
  Facts:
    companyName "CSIR",
    companyAddress csirAddress
  DifferentFrom: csirWithOtherAddress

Individual: csirWithOtherAddress
  Types: Company
  Facts:
    companyName "CSIR",
    companyAddress otherAddress
  DifferentFrom: csir

```

(7.7)

7.3 Non-delegated Class Cohesion

Formal Scenario Testing Technique: Inconsistent

7.3.1 Detection

Figure: 2.19, p. 25

OWL 2 Translation: Appendix D.1, p. 163

Non-delegated class cohesion is detected in a similar way in which multifaceted class cohesion has been detected. Setting up the equivalent scenario in Protégé (see listing (7.8)) as for the UML object diagram of Figure 2.20 (p. 26) does not lead to an inconsistency. Remember, the assumption is that a project may only have one project manager. Hence, it is desirable that the UML class diagram is modelled tightly enough such that if Sandy and Pablo work on the same project (i.e. ProjectX), it should not be possible for them to have different project managers.

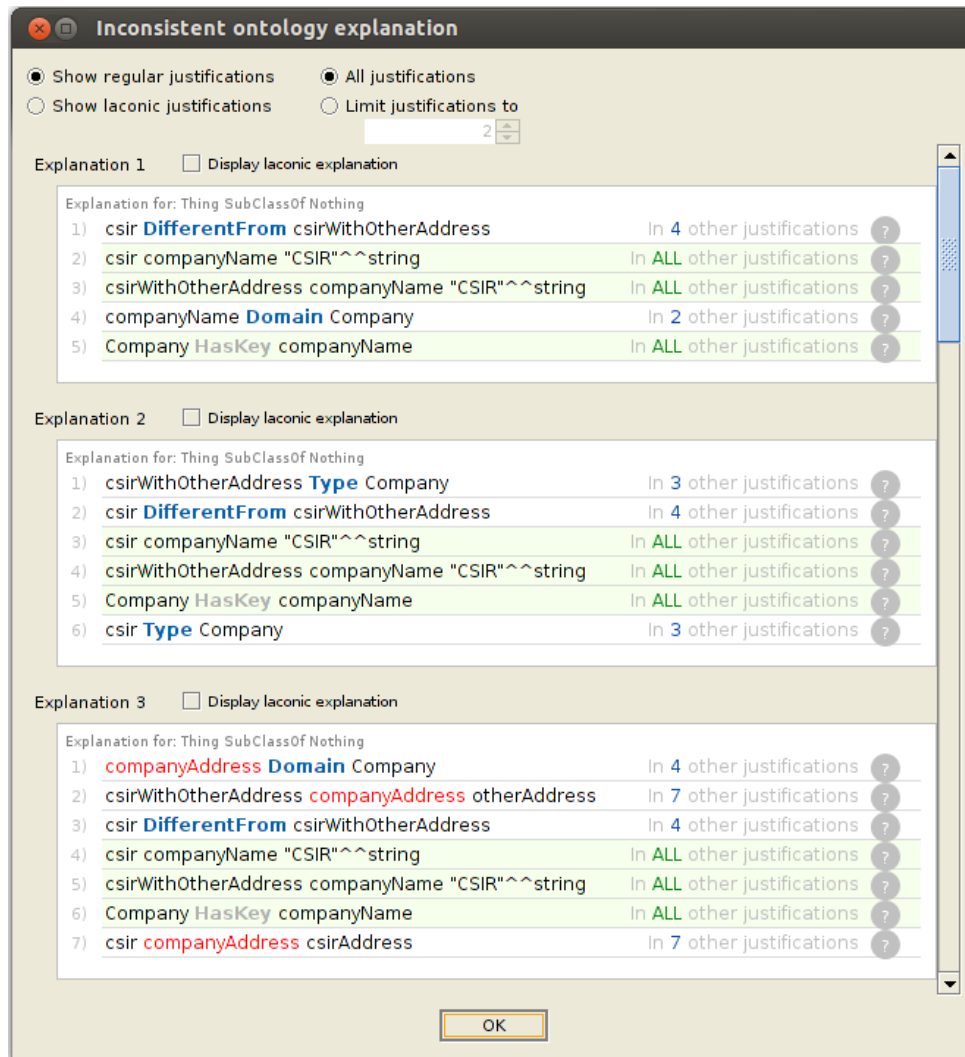


Figure 7.7: Multifaceted class cohesion validated by inconsistency.

```

Individual: sandy
Types: Employee
Facts:
    name "Sandy",
    dateOfBirth "1985-06-01",
    currentProject "ProjectX",
    projectManager "Ruth"
Individual: pablo
Types: Employee
Facts:
    name "Pablo",
    dateOfBirth "1968-04-11",
    currentProject "ProjectX",
    projectManager "Peet"

```

(7.8)

7.3.2 Validation

Figure: 2.21, p. 26

OWL 2 Translation: Appendix D.2, p. 164

When the scenario of listing (7.9) is applied on the OWL 2 translation of the redesigned UML class diagram, it results in the expected inconsistency. In this scenario, two individuals are added of type `Project`, each with the name `ProjectX`, but with different project managers. This is a scenario that should be disallowed and therefore this scenario must result in an inconsistency. The explanations for the inconsistency are given in Figure 7.8. This inconsistency confirms that the `Employee` and `Project` classes are free from non-delegated class cohesion.

```

Individual: projectX
Types: Project
Facts:
    name "ProjectX",
    projectManager "Ruth"
Individual: anotherProjectX
Types: Project
Facts:
    name "ProjectX",
    projectManager "Peet"

```

(7.9)

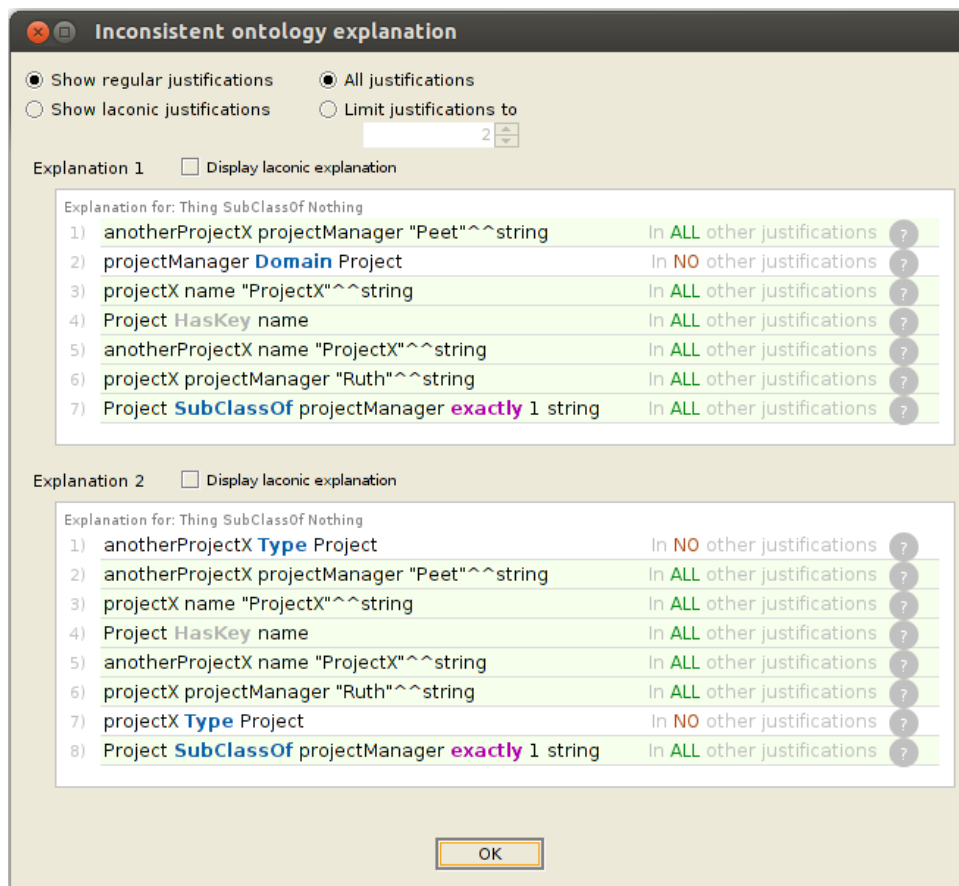


Figure 7.8: Non-delegated class cohesion validated by inconsistency.

7.4 Concealed Class Cohesion

Formal Scenario Testing Technique: Classification

Before concealed class cohesion can be detected, it is necessary to translate UML class diagrams in a way that will enable the detection of concealed concepts. The discussion on how to deal with qualified names in UML class diagrams (Section 5.7 on p. 76) refers. As stated in Section 5.7 (p. 76), when the same name is used for a feature in different UML classes, the relevant domains and ranges are translated as unions of the classes the feature appears in. To limit problems due to anonymous classes, explicitly named classes are introduced for domains and ranges that are expressed as unions of classes. Hence, for the `fromDate` attribute appearing in the classes `LeaveRequest` and `PerformanceReview`, the class `FromDateDomain` is introduced which is applied to the OWL 2 `fromDate` property as shown in listing (7.10).

```

Class: FromDateDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview
DataProperty: fromDate
  Domain: FromDateDomain
  Range: string

```

(7.10)

A further important point is that for the attribute `toDate`, also appearing in the classes `LeaveRequest` and `PerformanceReview`, the class `ToDateDomain` is introduced, which is also the union of the classes `LeaveRequest` and `PerformanceReview`, as was discussed in Section 5.7 (p. 76).

7.4.1 Detection

Figure: 2.23, p. 27

OWL 2 Translation: Appendix E.1, p. 165

As discussed in Section 2.2.4 (p. 27), concealed class cohesion is present in classes where a group of attributes/associations and/or operations are used together in more than one class. In order to detect concealed class cohesion, an individual of type `Thing` can be created. On this individual properties are set that correspond to attributes and/or operations that are suspected to appear in more than one class.

As an example, it will be shown how violations of concealed class cohesion can be detected for Figure 2.23 (p. 27). An individual `leaveRequest_calculatePeriodLength` of type `LeaveRequest_calculatePeriodLength_integer` is created to represent the operation `calculatePeriodLength` on the class `LeaveRequest`. Linking

`leaveRequest_calculatePeriodLength` to the individual `individualUsingPeriodInfo` via the `calculatePeriodLength` property states that the individual `individualUsingPeriodInfo` can perform the operation in question. Furthermore, the properties `fromDate` and `toDate` are asserted for the individual `individualUsingPeriodInfo`, which implies that the associated UML class have `fromDate` and `toDate` attributes/associations. The associated OWL 2 assertions are given in listing (7.11).

```

Individual: performanceReview_calculatePeriodLength
  Types:
    PerformanceReview_calculatePeriodLength_integer
Individual: leaveRequest_calculatePeriodLength
  Types:
    LeaveRequest_calculatePeriodLength_integer
Individual: individualUsingPeriodInfo
  Types: Thing
  Facts:
    calculatePeriodLength leaveRequest_calculatePeriodLength,
    fromDate "1972-09-10" string,
    toDate "2014-08-01" string

```

(7.11)

Running the reasoner on scenario (7.11) infers that `individualUsingPeriodInfo` is of type `CalculatePeriodLengthDomain`, `FromDateDomain` and `ToDateDomain` (see Figure 7.9). If the inferred class hierarchy is investigated, it is noted that the classes `CalculatePeriodLengthDomain`, `FromDateDomain` and `ToDateDomain` are all equivalent (see Figure 7.10 on p. 134). This gives some indication that the combination of attributes and operations, represented by the properties set on the individual, may define a concealed concept since the domains of these properties are equivalent.

What is confirmed is that an operation with name `calculatePeriodLength` appear on both the classes `PerformanceReview` and `LeaveRequest`. Whether these two operations have the same parameters and return type, is a question that still needs to be confirmed. This can be checked by creating an individual `calculatePeriodLengthOperation` of type `Thing` and assigning properties to it, which represent the signature of an operation as in listing (7.12) on p. 134. The property `calculatePeriodLength_inv` states that the operation is called on instances of class `LeaveRequest` while property `r_integer` states that its return type is of type `integer`. Since this operation does not take any parameters, no further properties need to be set to specify its signature. Running the reasoner on this ontology results in the inferences shown in Figure 7.11 (p. 135). It is inferred that the individual `calculatePeriodLengthOperation` is of type `CalculatePeriodLengthRange` and `R_integerDomain`. The inferred class hierarchy (Figure 7.12 on p. 136) shows that

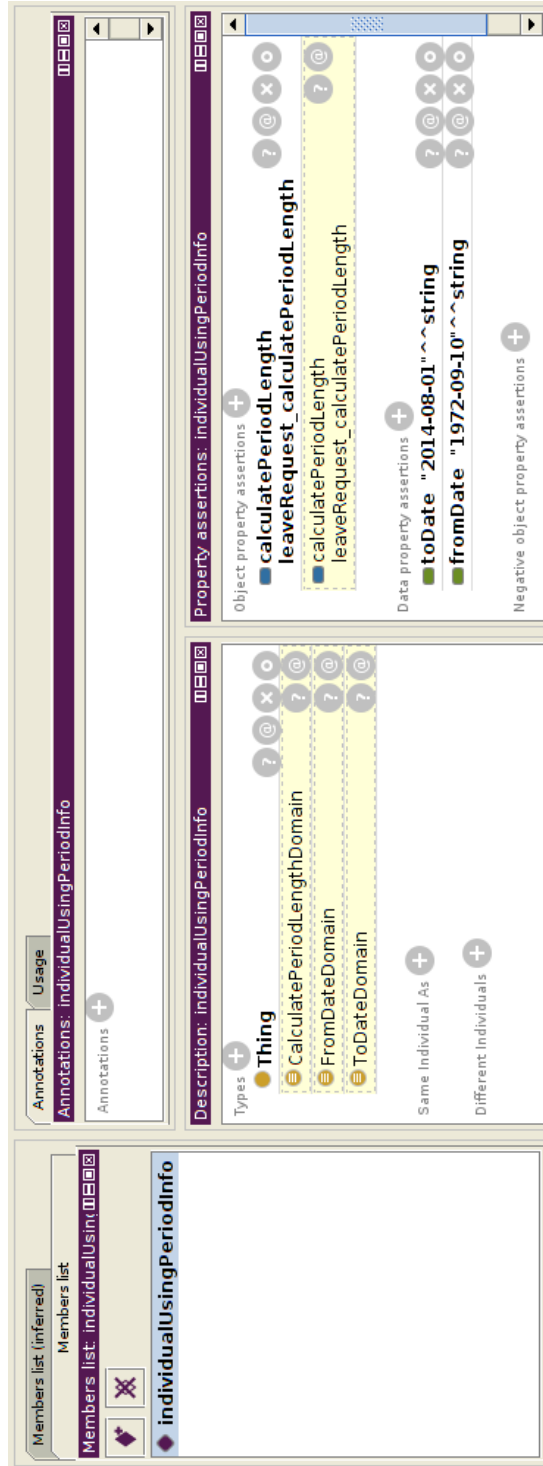


Figure 7.9: Classification of the individual `individualUsingPeriodInfo`.

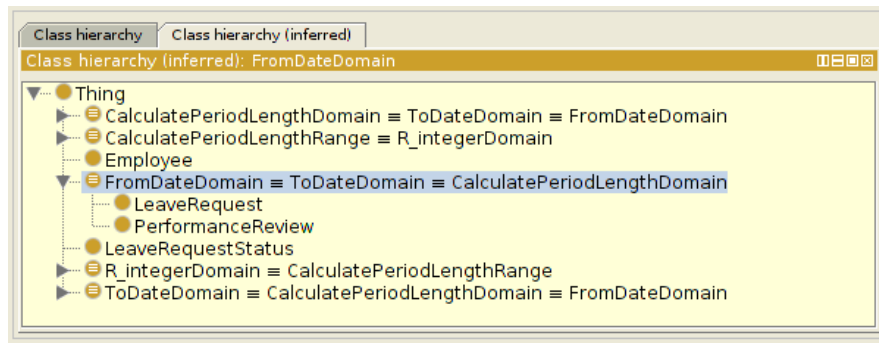


Figure 7.10: Inferred equivalences.

the classes `CalculatePeriodLengthRange` and `R_integerDomain` are equivalent as well as that they represent the classes `LeaveRequest_calculatePeriodLength` and `PerformanceReview_calculatePeriodLength`. Thus, it can be inferred that the operations `calculatePeriodLength` on the classes `LeaveRequest` and `PerformanceReview` have the same signatures.

```

Individual: leaveRequest
  Types: LeaveRequest
Individual: calculatePeriodLengthOperation
  Types: Thing
Facts:
  calculatePeriodLength_inv leaveRequest,
  r_integer 10 integer

```

(7.12)

In the scenarios (7.11) on p. 132 and (7.12), the operation `calculatePeriodLength():integer` on the class `LeaveRequest` has been considered. Note that similar inferences can be obtained by considering the same operation on the class `PerformanceReview`.

Based on the inferences in Figure 7.9 and the equivalences in Figure 7.10, it may seem as if it is only possible to detect concealed class cohesion when the domains of the properties in question are equivalent. Therefore, consider the case where the `fromDate` attribute/association appears on the classes `LeaveRequest`, `PerformanceReview` and `Employee`. Hence, the domain of the `fromDate` property is the union of the classes `LeaveRequest`, `PerformanceReview` and `Employee` as stated in (7.13) on p. 136.

The screenshot displays a software development tool interface with three main panels:

- Members list (inferred):** Shows members for `calculatePeriodLengthOperation` and `individualUsingPeriodInfo`.
- Annotations:** Shows annotations for `calculatePeriodLengthOperation`.
- Usage:** Shows usage information for `calculatePeriodLengthOperation`.

The **Annotations** panel is expanded, showing:

- Annotations:** `calculatePeriodLengthOperation`
- Description:** `calculatePeriodLengthOperation`
- Types:** `Thing`, `CalculatePeriodLengthRange`, `R_integerDomain`
- Same Individual As:** `Same Individual As`
- Different Individuals:** `Different Individuals`
- Property assertions:** `calculatePeriodLength_inv`, `leaveRequest`, `calculatePeriodLength_inv`, `leaveRequest`
- Data property assertions:** `r_integer_10`
- Negative object property assertions:** `Negative object property assertions`

Figure 7.11: Detecting the operations with the same signature.

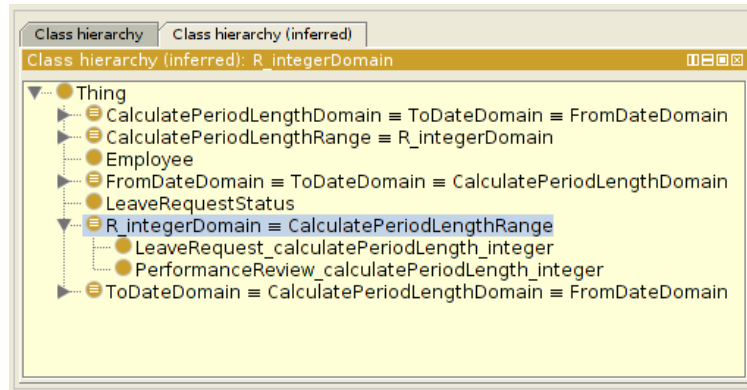


Figure 7.12: Inferred class hierarchy for operation.

```

Class: FromDateDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview or
    Employee
  DataProperty: fromDate
  Domain: FromDateDomain
  Range: string

```

(7.13)

If the same scenario as in (7.11) on p. 132 is applied, the inferences and inferred class hierarchy are obtained as shown in Figure 7.13 and Figure 7.14 (p. 138) respectively. The individual `individualUsingPeriodInfo` is classified as being of type `CalculatePeriodLengthDomain` and `ToDateDomain`. If the inferred class hierarchy is inspected again, it is noted that the class `FromDateDomain` subsumes the classes `CalculatePeriodLengthDomain`, `Employee` and `ToDateDomain`. This indicates, from a UML class diagram perspective, that the attributes `fromDate` and `toDate` along with the operation `calculatePeriodLength` appear together in the classes `LeaveRequest` and `PerformanceReview`. However, the attribute `fromDate` appears alone in the class `Employee`. Hence, a concealed concept is detected in the classes `LeaveRequest` and `PerformanceReview`, but not in the class `Employee`.

Figure 7.11 (p. 135) and Figure 7.12 (p. 136) have illustrated what inferences can be expected when the signatures of the operation `calculatePeriodLength` correspond in the classes `LeaveRequest` and `PerformanceReview`. Now consider instead that for the class `PerformanceReview` the operation `calculatePeriodLength` has the signature `calculatePeriodLength(newEmployee:boolean):integer`. If scenario (7.12) on p. 134 is added to this ontology, then the inferences as seen in Figure 7.11 and Figure 7.12 will still hold.

Scenario (7.14) can be created to test specifically for the new signature `calculatePeriodLength(newEmployee:boolean):integer` on class `PerformanceReview`. Running

The screenshot displays the IDE interface for the class `individualUsingPeriodInfo`. The top navigation bar includes 'Members list' and 'Members list (inferred)'. The main content area is divided into three sections:

- Members list:** Shows the class `individualUsingPeriodInfo` with a diamond icon.
- Annotations:** Contains 'Annotations' and 'Usage' tabs. The 'Annotations' tab is active, showing a list of annotations.
- Description: individualUsingPeriodInfo:** This section is divided into four sub-sections:
 - Types:** Lists 'Thing' as a type, highlighted with a yellow background.
 - Object property assertions:** Lists 'calculatePeriodLength' and 'leaveRequest_calculatePeriodLength'.
 - Data property assertions:** Lists 'toDate' and 'fromDate'.
 - Negative object property assertions:** Lists two entries for 'Negative object property assertions'.

Figure 7.13: Classification of the individual `individualUsingPeriodInfo` for the redesign.

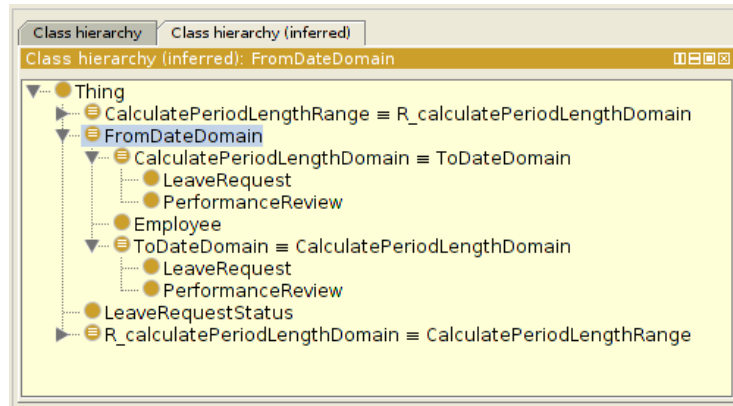


Figure 7.14: Inferred class hierarchy of the `FromDateDomain` class.

the reasoner on this scenario results in the inferences shown in Figure 7.15. It infers that the individual `performanceReview_calculatePeriodLength` is of type `PerformanceReview_calculatePeriodLength_integer` only. Consequently, this signature corresponds with only one operation in the UML class diagram.

```

Individual: performanceReview
Types: PerformanceReview
Individual: performanceReview_calculatePeriodLength
Types: Thing
Facts:
    calculatePeriodLength_inv performanceReview,
    newEmployee true boolean,
    r_integer 10 integer
  
```

(7.14)

7.4.2 Validation

Figure: 2.24, p. 27

OWL 2 Translation: Appendix E.2, p. 167

If scenario (7.15) on p. 141 is applied on the OWL 2 translation (see E.2 on p. 167) of the redesigned UML class diagram and the reasoner is run, the inferences as shown in Figure 7.16 (p. 140) are obtained. The individual `individualUsingPeriodInfo` is now classified as belonging to a single class named `Period`. Since `individualUsingPeriodInfo` is classified as belonging to a single class, it confirms that the combination of properties is used on a single class only.

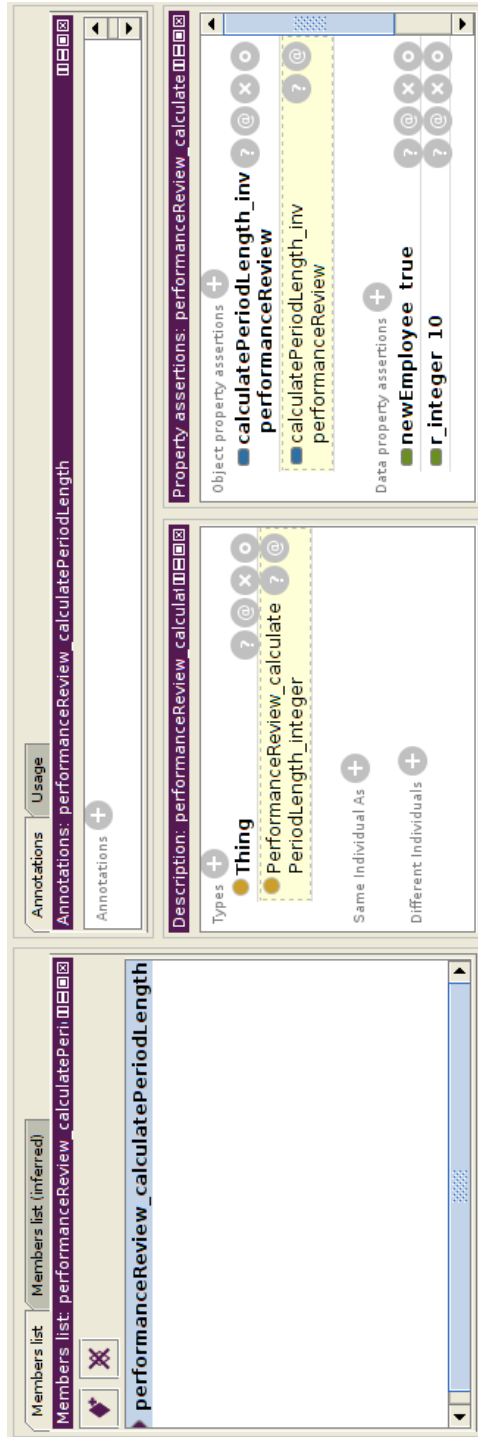


Figure 7.15: Validating the signature of an operation.

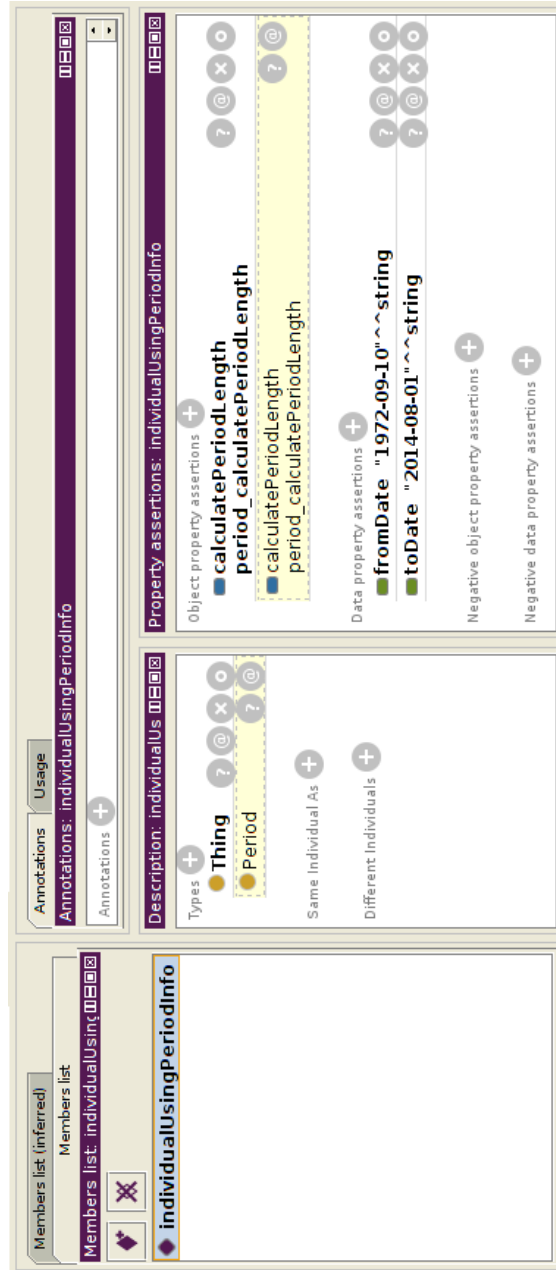


Figure 7.16: `individualUsingPeriodInfo` is classified as being of type `Period`.

```

Individual: period_calculatePeriodLength
  Types:
    Period_calculatePeriodLength_integer
Individual: individualUsingPeriodInfo
  Type: Thing
  Facts:
    calculatePeriodLength period_calculatePeriodLength,
    fromDate "1972-09-10" string,
    toDate "2014-08-01" string

```

(7.15)

7.5 Low Inheritance Cohesion of Attributes

Formal Scenario Testing Techniques: Classification, consistent

7.5.1 Detection

Figure: 2.25, p. 28

OWL 2 Translation: Appendix F.1, p. 170

In order to detect low inheritance cohesion, as seen in Figure 2.25 (p. 28), a classification scenario test is first applied. Listing (7.16) can be used to detect that it is not possible to assign properties to an individual of type `Thing` in such a way that it will ever be classified as being of type `Square`. The scenario (7.16) classifies the individual `square` as being of type `Rectangle` (see Figure 7.17 on p. 142).

```

Individual: square
  Type: Thing
  Facts:
    width 10 integer

```

(7.16)

Secondly, to indicate the problem with the conceptual schema of Figure 2.25 explicitly, the consistent scenario test as specified in (7.17) is used. In this scenario an individual `square` is created with its type set to `Square`. Moreover, the type is also set to make explicit that there is no need to set the `height` property on a `Square`. Running the reasoner results in the inconsistency displayed in Figure 7.18.

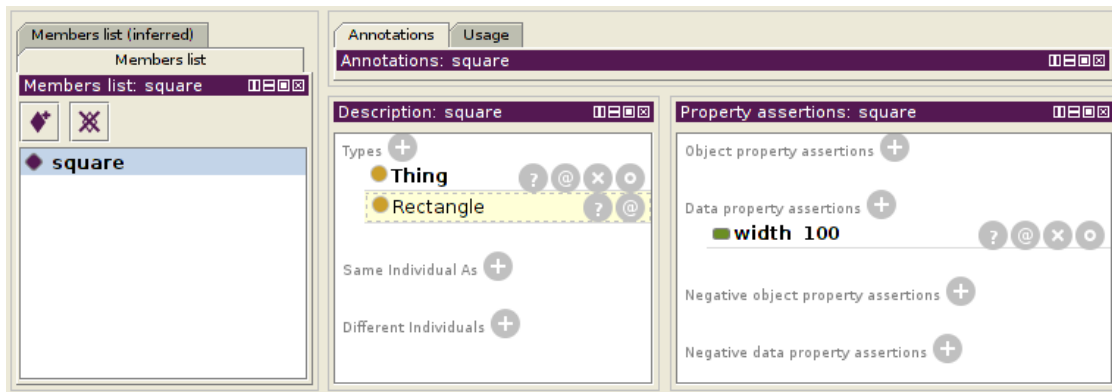


Figure 7.17: The individual `square` is classified as being of type `Rectangle`.

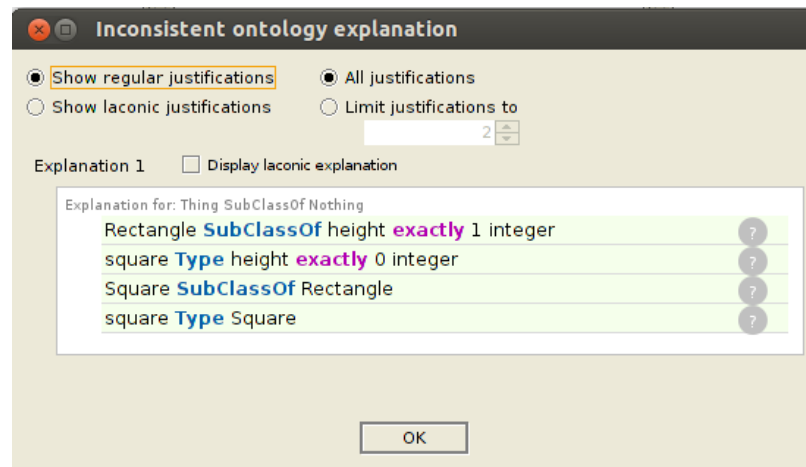


Figure 7.18: Explanation of the inconsistency for `square`.

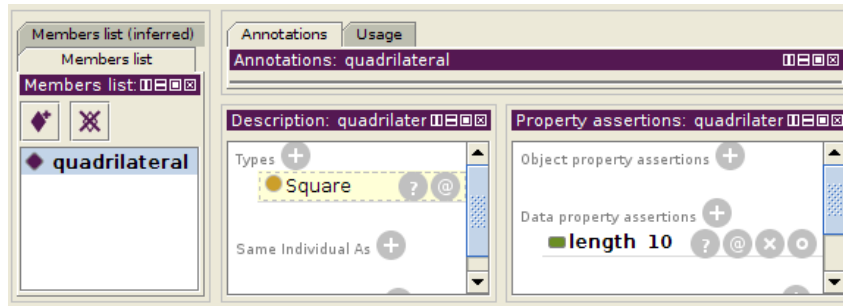
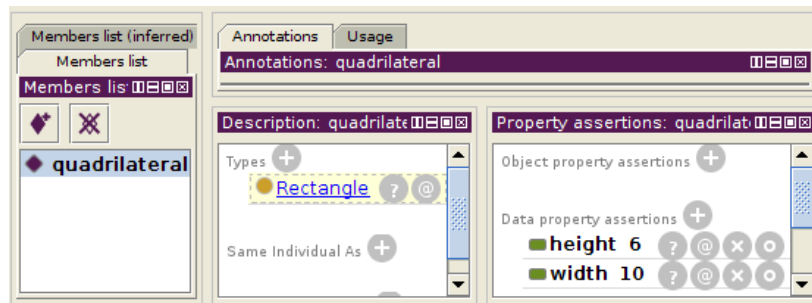
Individual: `square`
 Type:
 `Square`,
 `height exactly 0 integer`

(7.17)

7.5.2 Validation

Figure: 2.26, p. 28

OWL 2 Translation: Appendix F.2, p. 170

Figure 7.19: Correct classification of the individual `quadrilateral`.Figure 7.20: Individual with `height` and `width` properties is classified as a `Rectangle`.

```

Individual: quadrilateral
Type: Thing
Facts:
    length 10 integer
  
```

(7.18)

```

Individual: quadrilateral
Type: Thing
Facts:
    height 6 integer
    width 10 integer
  
```

(7.19)

If the scenario (7.18) is applied to the OWL 2 translation of the redesigned UML class diagram (see Appendix F.2 on p. 170), the individual `quadrilateral` is classified as being of type `Square` (see Figure 7.19). If both `height` and `width` properties are added to the `quadrilateral` individual in scenario (7.19), it is classified as a `Rectangle` as shown in Figure 7.20. For the redesign, the scenario (7.17) is now consistent. These formal scenario tests confirm that the `Square` and `Rectangle` UML class diagram has been improved.

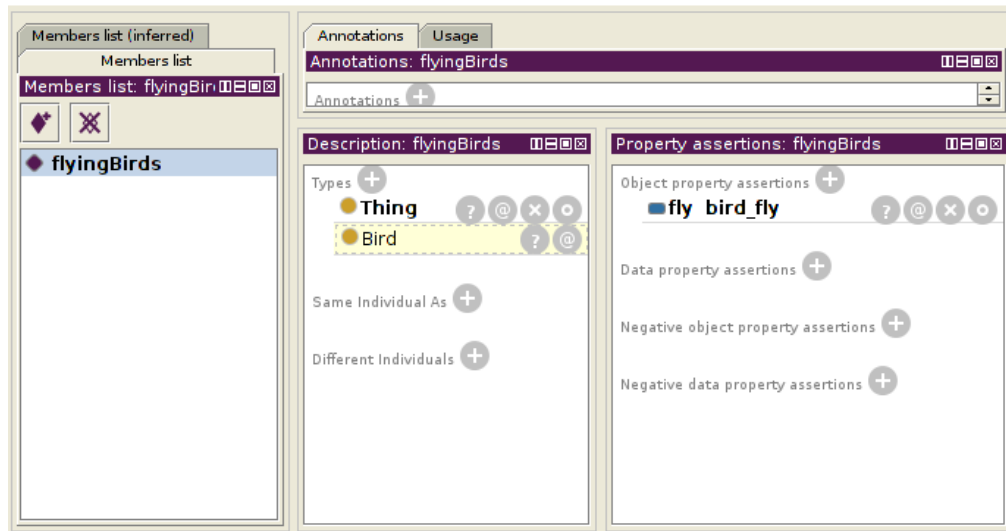


Figure 7.21: Classification based on the `fly()` operation yields the class `Bird`.

7.6 Low Inheritance Cohesion of Operations

Formal Scenario Testing Techniques: Classification, consistent

7.6.1 Detection

Figure: 2.27, p. 29

OWL 2 Translation: Appendix F.3, p. 29

Furthermore, the techniques defined here are applied on UML class diagrams and not on code as is the case with approach of Etzkorn, et. al.

A classification scenario test can be used to determine all the birds that can fly. This scenario is listed in listing (7.20). To achieve this, an individual `bird_fly` is created of type `Bird_fly` which represents the `fly()` operation. To represent flying birds, an individual `flyingBirds` of type `Thing` is created to which the property `fly` is assigned. Running the reasoner results in `flyingBirds` being classified as `Bird` (Figure 7.21). Consulting the inferred class hierarchy (as seen in Figure 7.22) shows that the class `Bird` subsumes the classes `Penguin` and `Eagle`. This is incorrect since penguins cannot fly.

```

Individual: bird_fly
  Type: Bird_fly
Individual: flyingBirds
  Type: Thing
  Facts: fly bird_fly
  
```

(7.20)

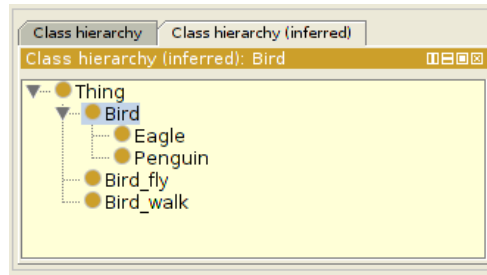


Figure 7.22: The Bird class includes both the Penguin and Eagle classes.

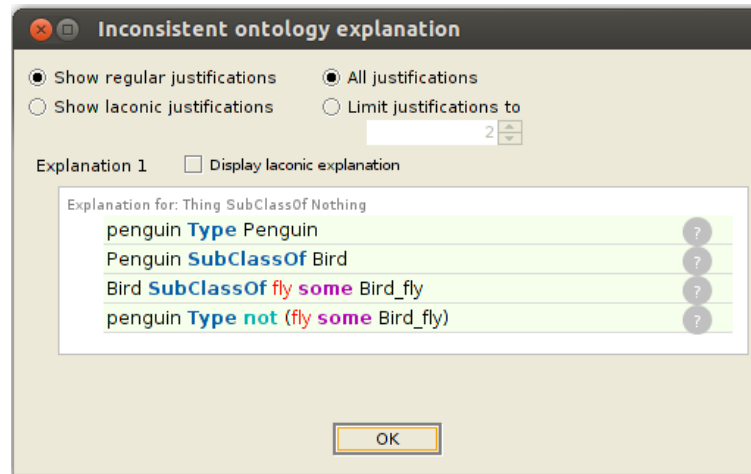


Figure 7.23: Penguins not flying results in an inconsistency.

By applying a consistent scenario test, the problem with the UML class diagram of Figure 2.27 (p. 29) can be made more apparent. Again, in listing (7.21) an individual `bird_fly` of type `Bird_fly` is defined to represent a `fly()` operation. An individual of type `Penguin` is created and the believe that penguins are not suppose to fly are made explicit by adding the type `not (fly some bird_fly)`. Running the reasoner on this scenario results in the inconsistency shown in Figure 7.23.

```

Individual: bird_fly
  Type: Bird_fly
Individual: penguin
  Type: Penguin,
      not (fly some bird_fly)
  
```

(7.21)

7.6.2 Validation

Figure: 2.28, p. 29

OWL 2 Translation: Appendix F.4, p. 172

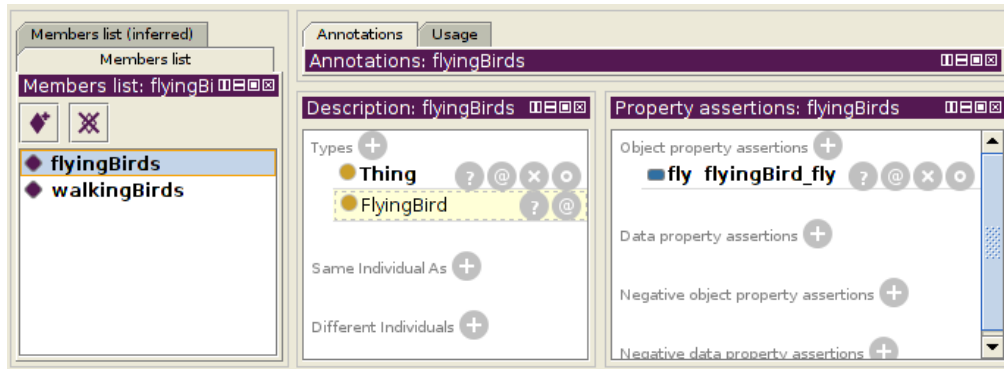


Figure 7.24: Only birds of type FlyingBird can fly.

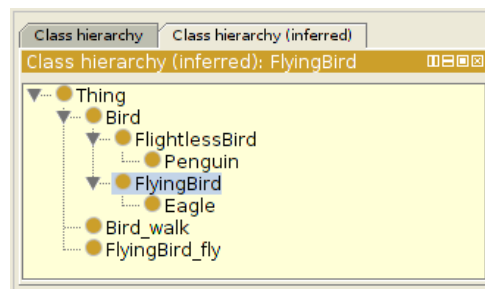


Figure 7.25: Eagles are the only birds of type FlyingBird.

The OWL 2 translation for Figure 2.28 (p. 29) is given in Appendix F.4 (p. 172). The scenario in (7.20) on p. 144 is rewritten as as (7.22). This results in the classification and inferred class hierarchy as seen in Figure 7.24 and 7.25 respectively. The classification indicates that only birds of type `FlyingBird` can fly and from the inferred class hierarchy it follows that only eagles are of type `FlyingBird`.

```

Individual: flyingBird_fly
  Type: FlyingBird_fly
Individual: flyingBirds
  Type: Thing
  Facts: fly flyingBird_fly
  
```

(7.22)

For completeness sake the scenario in (7.22) can be adjusted to classify all birds that can walk. Figures 7.26 and 7.27 confirm the expectation that both penguins and eagles can walk.

Rewriting scenario (7.21) on p. 145 as (7.23) and running the reasoner over it is now consistent. This illustrates that the UML class diagram of Figure 2.28 is now in agreement with the general understanding of eagles and penguins.

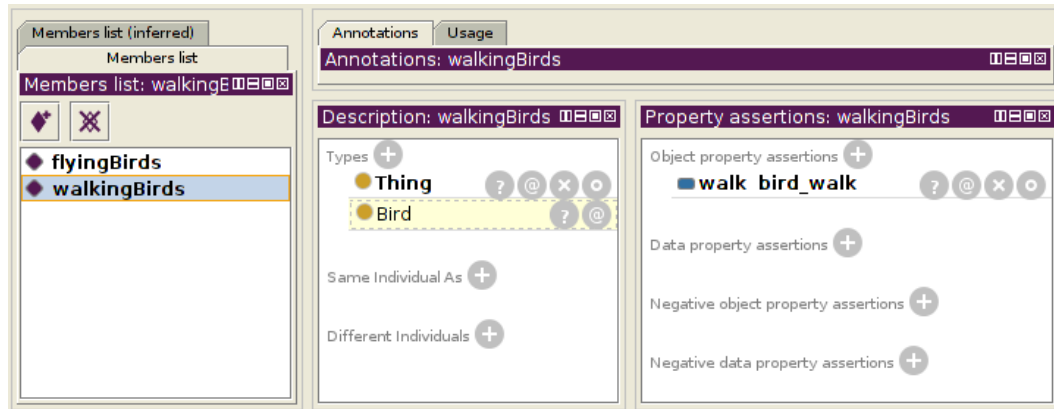


Figure 7.26: All things of type Bird can walk.

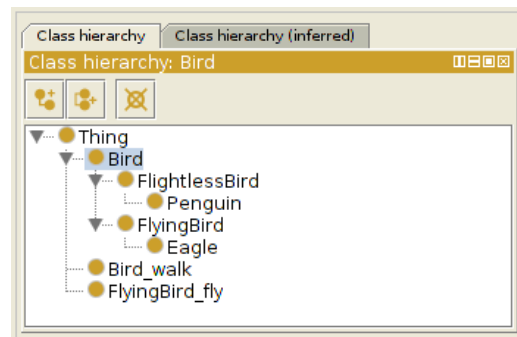


Figure 7.27: Both eagles and penguins can walk.

```

Individual: flyingBird_fly
  Type: FlyingBird_fly
Individual: penguin
  Type: Penguin,
  not (fly some flyingBird_fly)

```

(7.23)

7.7 Contribution and Related Research

This section gives a synopsis of the contributions of this chapter (see Section 7.7.1) with related research being discussed in Section 7.7.2 (p. 148).

7.7.1 Contribution

The contributions of this chapter builds on the work of Eder, et. al. [38]. Briand, et. al. have levelled the following criticism against the work of Eder, et. al. [21]:

“Within this framework, an analysis of the semantics of a given method or class is required to determine its degree of method, class or inheritance cohesion. Such an analysis cannot be automated. If we use this framework to derive cohesion

Table 7.1: Formal scenario testing techniques used for detection and verification of heuristics

Heuristic violation	Formal scenario testing technique(s)	Reference
Separable class cohesion	Classification scenario test	7.1, p. 120
Multifaceted class cohesion	Inconsistent scenario test	7.2, p. 125
Non-delegated class cohesion	Inconsistent scenario test	7.3, p. 127
Concealed class cohesion	Classification scenario test	7.4, p. 131
Low inheritance cohesion	Classification and consistent scenario tests	7.5, p.141 and 7.6, p. 144

measures, the resulting measures will not be automatically collectable. This is a severe impediment to their widespread use.”

The contribution of this chapter is to present techniques to detect and validate the cohesion heuristics defined by Eder, et. al. [38] in a semi-automated fashion. A summary is given of the heuristics and the related formal scenario testing techniques that are used to detect and validate violations in Table 7.1.

Naturally these heuristics can be detected and corrected through manual means. However, conceptual schemas in practice are often large and created and maintained over a long period of time, which makes detecting these heuristics manually difficult and error-prone. Furthermore, there is substantial variance in modeller expertise in practice, which may result in modellers having difficulty reasoning about normal forms, as is required for detecting multifaceted- and non-delegated class cohesion violations [79, 111]. Various database design tools are available to support database normalization [37], but no tools exist for facilitating the design process for UML class diagrams. The semi-automated approach discussed in this section, takes the first tentative steps which could eventually result in providing tool support for the design of UML class diagrams.

Finally, even though this chapter discussed the various means through which cohesion related heuristics can be detected and validated, it is worthwhile to bear in mind that none of these techniques can give indisputable proof of the conceptual schema being correct or incorrect. However, equipped with the techniques discussed here, a modeller can gain greater insight into the appropriateness or inappropriateness of a conceptual schema.

7.7.2 Related Research

Most of the work done regarding object-oriented heuristics is based on syntactical analysis of object-oriented code bases. A detailed discussion in this regard is provided by Lanza, et. al. [81].

An automated approach to the semantic analysis of code is presented by Etkorn, et. al. [39]. Their approach employs a knowledge-based system which is equipped with a natural

language program understanding system. It parses object oriented code bases and performs natural language understanding on comments and identifiers found in the code, from which concepts are extracted which are stored as conceptual graphs in the knowledge-base. Semantic analysis of a class involves understanding each method separately and storing a conceptual graph in the knowledge base that is representative of the meaning of the method. They introduce a metric, called the Logical Related of Methods (LORM), which gives an indication of how related the concepts are for a pair of methods. Further key metrics that are provided are a metric for measuring the complexity of a class and the overlap in functionality for a pair of classes.

The main differences between the research presented in this chapter and that of Etkorn, et. al. are that the current research makes no attempt to provide metrics to measure semantics or to provide full automation. Rather, it provides a set of techniques which can be applied by a modeller in order to detect classification violations. Furthermore, the techniques defined here are applied on UML class diagrams and not on code, as is the case with approach of Etkorn, et. al.

Chapter 8

Conclusion

In this chapter a concise synopsis is given of the contributions of this dissertation (see Section 8.1) and in Section 8.2 (p. 152) avenues for further research are discussed.

8.1 Contribution

The contributions of this dissertation are clarified by explaining

1. the gap in existing research (see the next section),
2. how this gap is addressed by the contributions of this dissertation (see Section 8.1.2 on p. 150) and
3. the value proposition of the research presented here (see Section 8.1.3 on p. 151).

In Section 8.1.4 (p. 151) presentations and publications that form part of this research effort are included.

8.1.1 The Research Gap Identified

As mentioned in Section 1.4.2 (p. 9) and illustrated in Chapter 6 (p. 92), existing DL translations of UML class diagrams are aimed at verification rather than validation. That is, existing DL translations of UML class diagrams can detect logical inconsistencies in UML class diagrams, but they cannot comment on whether the UML class diagram is a cohesive and an accurate representation of the business requirements.

8.1.2 How the Gap is addressed by this Research

Formal scenario testing is an attempt to at least in part address the gap as defined in the previous section. The high-level contributions with regards to formal scenario testing are listed below with references to where the detailed contributions can be found.

1. The formal scenario testing approach and related techniques and guidelines are defined in Chapter 6 (p. 92). Feedback is also given on a small case study where formal scenario testing has been used on a real-world project.

2. Formal scenario testing builds on existing research which translates UML class diagrams to DLs and OWL 2. However, not all UML class diagram features (that are of importance to formal scenario testing) have been translated to DLs and OWL 2. Also, existing translations are not always sufficient for the requirements of formal scenario testing. In Chapter 5 (p. 60) existing UML to DL/OWL 2 translations are extended for the purpose of formal scenario testing.
3. In Chapter 7 (p. 119) it is illustrated how formal scenario testing can be applied to give evidence of the cohesiveness, or lack of cohesiveness, of the classes and class hierarchies found in a UML class diagram.

8.1.3 The Value Proposition of this Research

Throughout this dissertation, the focus has been on the requirements engineering phase of the SDLC (see Section 1.1 on p. 1). As stated in Chapter 1 (p. 1), arriving at a complete and consistent software specification is one of the most challenging tasks in software engineering. Moreover, detecting and correcting errors during the requirements engineering phase is 100 times cheaper than when the software is in production. Existing DL translations of UML class diagrams contribute to this effort by enabling formal verification of UML class diagrams. Formal scenario testing extends this effort by facilitating formal validation of UML class diagrams. Chapter 6 (p. 92) introduced formal scenario testing and showed how it can be used to validate UML class diagrams. Chapter 7 (p. 119) showed how formal scenario testing can be used to test various object-oriented cohesion heuristics.

What has not been made explicit is that formal scenario testing can be applied to ontology engineering. Indeed, Falbo, et. al. [41] observe that in ontology engineering the ontology development process should be quite similar to that of the software development process of software engineering. In a similar vein as for software engineering, can formal scenario testing be utilized in ontology engineering to gather evidence of whether an ontology is in agreement with the business requirements or not.

8.1.4 Presentation and Publication in Support of this Dissertation

The work in this dissertation is supported by the following presentation and publication:

Presentation [50] H. Harmse, A. Britz, and A. Gerber. Scenario Testing on UML Class Diagrams using Description Logics. "<http://www.cair.za.net/research/outputs/scenario-testing-uml-class-diagrams-using-description-logic>", 2013.

Publication [51] H. Harmse, K. Britz, A. Gerber, and D. Moodley. Scenario testing using formal ontologies. In G. Guizzardi, O. Pastor, Y. Wand, S. D. Cesare, F. Gailly, M. Lycett, and C. Partridge, editors, *1st Joint Workshop ONTO.COM / ODISE on*

Ontologies in Conceptual Modeling and Information Systems Engineering, volume 1301 of CEUR Workshop Proceedings. CEUR Workshop Proceedings, 2014. CEUR-WS.org, 2014.

8.2 Future Research

In Section 6.2 (p. 94) techniques have been defined for constructing formal scenario tests in an ad hoc fashion. If a generic methodology can be defined for constructing these scenario tests, it will pave the way towards (semi-)automation and the eventual creation of tools in support of formal scenario testing.

The research presented in this dissertation has focussed on the creation of a complete and consistent UML class diagram that is an accurate representation of the business requirements. Constraints expressed in the Object Constraint Language (OCL) [3] have been explicitly excluded from this dissertation due to scope constraints. The usefulness of OCL in enriching the semantics of UML class diagrams is undeniable. It can be used to define constraints [3] such as class invariants, which are constraints that apply to every class instance as a whole [89], and pre- and post conditions on operations, which are required to model the functionality of a software system [116]. However, it is well-known that OCL is undecidable. Queralt, et. al. [100] have defined a fragment of OCL, called OCL-lite, which is decidable and which guarantees finite satisfiability. Investigating how OCL-lite can be incorporated into formal scenario testing will improve the usefulness of formal scenario testing.

Section 6.4 (p. 108) discussed a small case study in which formal scenario testing have been used on a real-world project. In order to understand the potential benefits and shortcomings of the formal scenario testing approach described here, it is imperative to conduct larger scale case studies. The feedback from these case studies will be vital in fine-tuning formal scenario testing for maximum benefit.

Section 2.2 (p. 21) explained how ideas from data normalization can be applied to improve class cohesion. From a class cohesion perspective only second and third normal forms have been considered in the literature [38]. The question is: “Can class cohesion benefit from taking into consideration higher normal forms?”

8.3 Summary

This chapter gave a brief overview of the research gap that was identified and explained at a high-level how formal scenario testing, the main contribution of this dissertation, addresses this gap.

Appendices

Appendix A

RatesConfig Class Diagram Translated to OWL 2

Here we provide the complete translation of the UML class diagram of Figure 6.17 to OWL 2 Manchester Syntax [57].

```
Class: ChargeType
  DisjointUnionOf: HotelChargeType,
  PAXChargeType, RoomTypeChargeType
Class: CriterionType
  DisjointUnionOf: InterleadingCriterionType,
  BlockBookingCriterionType, ChannelCriterionType
Class: RateConfig
  SubClassOf:
    chargeType exactly 1 Thing,
    criterionType exactly 1 Thing
  DisjointUnionOf: InterleadingHotelRateConfig,
  BlockBookingHotelRateConfig, ChannelHotelRateConfig,
  ChannelPAXRateConfig, ChannelRoomTypeRateConfig,
  BlockBookingRoomTypeRateConfig

ObjectProperty: chargeType
  Domain: RateConfig
  Range: ChargeType

ObjectProperty: criterionType
  Domain: RateConfig
  Range: CriterionType
Class: HotelChargeType
  SubClassOf: ChargeType
Class: PAXChargeType
  SubClassOf: ChargeType
Class: RoomTypeChargeType
  SubClassOf: ChargeType
```

```
Class: InterleadingCriterionType
  SubClassOf: CriterionType
Class: BlockBookingCriterionType
  SubClassOf: CriterionType
Class: ChannelCriterionType
  SubClassOf: CriterionType

Class: InterleadingHotelRateConfig
  SubClassOf: RateConfig,
    hotelChargeType exactly 1 Thing,
    interleadingCriterionType exactly 1 Thing
Class: BlockBookingHotelRateConfig
  SubClassOf: RateConfig,
    hotelChargeType exactly 1 Thing,
    blockBookingCriterionType exactly 1 Thing
Class: ChannelHotelRateConfig
  SubClassOf: RateConfig,
    hotelChargeType exactly 1 Thing,
    channelCriterionType exactly 1 Thing
Class: ChannelPAXRateConfig
  SubClassOf: RateConfig,
    paxChargeType exactly 1 Thing,
    channelCriterionType exactly 1 Thing
Class: ChannelRoomTypeRateConfig
  SubClassOf: RateConfig,
    roomTypeChargeType exactly 1 Thing,
    channelCriterionType exactly 1 Thing
Class: BlockBookingRoomTypeRateConfig
  SubClassOf: RateConfig,
    roomTypeChargeType exactly 1 Thing,
    blockBookingCriterionType exactly 1 Thing

ObjectProperty: hotelChargeType
  SubPropertyOf: chargeType
  Domain: BlockBookingHotelRateConfig
    or ChannelHotelRateConfig
    or InterleadingHotelRateConfig
  Range: HotelChargeType
```

ObjectProperty: paxChargeType
SubPropertyOf: chargeType
Domain: ChannelPAXRateConfig
Range: PAXChargeType

ObjectProperty: roomTypeChargeType
SubPropertyOf: chargeType
Domain: BlockBookingRoomTypeRateConfig
or ChannelRoomTypeRateConfig
Range: RoomTypeChargeType

ObjectProperty: channelCriterionType
SubPropertyOf: criterionType
Domain: ChannelHotelRateConfig
or ChannelPAXRateConfig
or ChannelRoomTypeRateConfig
Range: ChannelCriterionType

ObjectProperty: interleadingCriterionType
SubPropertyOf: criterionType
Domain: InterleadingHotelRateConfig
Range: InterleadingCriterionType

ObjectProperty: blockBookingCriterionType
SubPropertyOf: criterionType
Domain: BlockBookingHotelRateConfig
or BlockBookingRoomTypeRateConfig
Range: BlockBookingCriterionType

DisjointProperties: hotelChargeType,
paxChargeType, roomTypeChargeType

DisjointProperties: blockBookingCriterionType,
channelCriterionType, interleadingCriterionType

Appendix B

Separable Class Cohesion Examples Translated to OWL 2

B.1 Translation of Employee Class with Separable Class Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.11 to OWL 2 Manchester Syntax [57].

```
DataProperty: employeeCode
  Domain: Employee
  Range: string
DataProperty: employeeName
  Domain: Employee
  Range: string
DataProperty: salary
  Domain: Employee
  Range: integer

DataProperty: projectName
  Domain: Employee
  Range: string
DataProperty: projectCost1
  Domain: Employee
  Range: integer
DataProperty: projectCost2
  Domain: Employee
  Range: integer

ObjectProperty: calculateSalaryIncrease_inv
  Domain: Employee_calculateSalaryIncrease_integer
  Range: Employee
```

```
ObjectProperty: calculateSalaryIncrease
  InverseOf: calculateSalaryIncrease_inv
DataProperty: increase
  Domain: Employee_calculateSalaryIncrease_integer
  Range: integer

ObjectProperty: calculateProjectCost_inv
  Domain: Employee_calculateProjectCost_integer
  Range: Employee
ObjectProperty: calculateProjectCost
  InverseOf: calculateProjectCost_inv
DataProperty: r_integer
  Domain: Employee_calculateSalaryIncrease_integer
  or Employee_calculateProjectCost_integer
  Range: integer

Class: Employee
  SubClassOf:
    employeeCode exactly 1 string,
    employeeName exactly 1 string,
    salary exactly 1 integer,
    projectName exactly 1 string,
    projectCost1 exactly 1 integer,
    projectCost2 exactly 1 integer,
    calculateProjectCost some Employee_calculateProjectCost_integer,
    calculateSalaryIncrease some Employee_calculateSalaryIncrease_integer

Class: Employee_calculateSalaryIncrease_integer
  SubClassOf:
    calculateSalaryIncrease_inv exactly 1 Thing,
    increase exactly 1 integer,
    r_integer exactly 1 integer
  HasKey:
    calculateSalaryIncrease_inv, increase

Class: Employee_calculateProjectCost_integer
  SubClassOf:
    calculateProjectCost_inv exactly 1 Thing,
    r_integer exactly 1 integer
  HasKey:
    calculateProjectCost_inv
```

B.2 Translation of Redesigned Employee Class

Here we provide the complete translation of the UML class diagram of Figure 2.13 to OWL 2 Manchester Syntax [57].

```
DataProperty: employeeCode
  Domain: Employee
  Range: string
DataProperty: employeeName
  Domain: Employee
  Range: string
DataProperty: salary
  Domain: Employee
  Range: integer

DataProperty: projectName
  Domain: Project
  Range: string
DataProperty: projectCost1
  Domain: Project
  Range: integer
DataProperty: projectCost2
  Domain: Project
  Range: integer

ObjectProperty: calculateSalaryIncrease_inv
  Domain: Employee_calculateSalaryIncrease_integer
  Range: Employee
ObjectProperty: calculateSalaryIncrease
  InverseOf: calculateSalaryIncrease_inv
DataProperty: increase
  Domain: Employee_calculateSalaryIncrease_integer
  Range: integer
```


ObjectProperty: calculateProjectCost_inv
Domain: Project_calculateProjectCost_integer
Range: Project
ObjectProperty: calculateProjectCost
InverseOf: calculateProjectCost_inv
DataProperty: r_integer
Domain: Employee_calculateSalaryIncrease_integer
or Project_calculateProjectCost_integer
Range: integer

Class: Employee
SubClassOf:
employeeCode max 1 string,
employeeName max 1 string,
salary max 1 integer,
calculateSalaryIncrease some Employee_calculateSalaryIncrease_integer
DisjointWith: Project

Class: Project
SubClassOf:
projectName max 1 string,
projectCost1 max 1 integer,
projectCost2 max 1 integer,
calculateProjectCost some Project_calculateProjectCost_integer
DisjointWith: Employee

Class: Employee_calculateSalaryIncrease_integer
SubClassOf:
calculateSalaryIncrease_inv exactly 1 Thing,
increase exactly 1 integer,
r_integer exactly 1 integer
HasKey:
calculateSalaryIncrease_inv, increase

Class: Project_calculateProjectCost_integer
SubClassOf:
calculateProjectCost_inv exactly 1 Thing,
r_integer exactly 1 integer
HasKey:
calculateProjectCost_inv

Appendix C

Multifaceted Class Cohesion Examples Translated to OWL 2

C.1 Translation of ContactInformation Class with Multifaceted Class Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.15 to OWL 2 Manchester Syntax [57].

```
ObjectProperty: companyAddress
  Domain: ContactInformation
  Range: Address
DataProperty: companyName
  Domain: ContactInformation
  Range: string
DataProperty: phoneNumber
  Domain: ContactInformation
  Range: string
DataProperty: contactPerson
  Domain: ContactInformation
  Range: string

Class: ContactInformation
  SubClassOf:
    companyName exactly 1 string,
    contactPerson exactly 1 string,
    companyAddress exactly 1 Thing,
    phoneNumber exactly 1 string
  HasKey:
    companyName, contactPerson
  DisjointWith:
    Address
```

```

Class: Address
  DisjointWith:
    ContactInformation

```

C.2 Translation of Redesigned ContactInformation Class

Here we provide the complete translation of the UML class diagram of Figure 2.17 to OWL 2 Manchester Syntax [57].

```

ObjectProperty: companyAddress
  Domain: Company
  Range: Address
ObjectProperty: company
  Domain: ContactInformation
  Range: Company
DataProperty: companyName
  Domain: Company
  Range: string
DataProperty: phoneNumber
  Domain: ContactInformation
  Range: string
DataProperty: contactPerson
  Domain: ContactInformation
  Range: string

Class: Address
Class: Company
  SubClassOf:
    companyName exactly 1 string,
    companyAddress exactly 1 Thing
  HasKey:
    companyName
Class: ContactInformation
  SubClassOf:
    company exactly 1 Thing,
    contactPerson exactly 1 string,
    phoneNumber exactly 1 string,
  HasKey:
    company, contactPerson
DisjointClasses:
  Address, Company, ContactInformation

```

Appendix D

Non-delegated Class Cohesion Examples Translated to OWL 2

D.1 Translation of Employee Class with Non-delegated Class Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.19 to OWL 2 Manchester Syntax [57].

```
DataProperty: name
  Domain: Employee
  Range: string
DataProperty: dateOfBirth
  Domain: Employee
  Range: string
DataProperty: currentProject
  Domain: Employee
  Range: string
DataProperty: projectManager
  Domain: Employee
  Range: string

Class: Employee
  SubClassOf:
    name exactly 1 string,
    dateOfBirth exactly 1 string,
    currentProject exactly 1 string,
    projectManager exactly 1 string
  HasKey:
    name
```

D.2 Translation of Redesigned Employee Class

Here we provide the complete translation of the UML class diagram of Figure 2.21 to OWL 2 Manchester Syntax [57].

```
DataProperty: name
  Domain: NameDomain
  Range: string
DataProperty: dateOfBirth
  Domain: Employee
  Range: string
DataProperty: currentProject
  Domain: Employee
  Range: string
DataProperty: projectManager
  Domain: Project
  Range: string
ObjectProperty: currentProject
  Domain: Employee
  Range: Project

Class: NameDomain
  EquivalentTo:
    Employee or Project
Class: Employee
  SubClassOf:
    name exactly 1 string,
    dateOfBirth exactly 1 string,
    currentProject exactly 1 Thing
  HasKey:
    name
  DisjointWith:
    Project
Class: Project
  SubClassOf:
    name exactly 1 string,
    projectManager exactly 1 string
  HasKey:
    name
  DisjointWith:
    Employee
```

Appendix E

Concealed Class Cohesion Examples Translated to OWL 2

E.1 Translation of LeaveRequest and PerformanceReview Classes with Concealed Class Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.23 to OWL 2 Manchester Syntax [57].

```
Class: PerformanceReview
  SubClassOf:
    employee exactly 1 Thing,
    fromDate exactly 1 string,
    toDate exactly 1 string,
    manager exactly 1 Thing,
    calculatePeriodLength
      some PerformanceReview_calculatePeriodLength_integer
  HasKey:
    employee, fromDate, toDate
Class: LeaveRequest
  SubClassOf:
    id exactly 1 integer,
    reason exactly 1 string,
    fromDate exactly 1 string,
    toDate exactly 1 string,
    status exactly 1 Thing,
    calculatePeriodLength
      some LeaveRequest_calculatePeriodLength_integer
  HasKey:
    id
Class: Employee
Class: LeaveRequestStatus
```

```
ObjectProperty: employee
  Domain: PerformanceReview
  Range: Employee
ObjectProperty: manager
  Domain: PerformanceReview
  Range: Employee

DataProperty: fromDate
  Domain: FromDateDomain
  Range: string
DataProperty: toDate
  Domain: ToDateDomain
  Range: string

DataProperty: id
  Domain: LeaveRequest
  Range: integer
DataProperty: reason
  Domain: LeaveRequest
  Range: string
ObjectProperty: status
  Domain: LeaveRequest
  Range: LeaveRequestStatus

Class: CalculatePeriodLengthDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview
Class: CalculatePeriodLengthRange
  EquivalentTo:
    LeaveRequest_calculatePeriodLength_integer or
    PerformanceReview_calculatePeriodLength_integer
Class: FromDateDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview
Class: ToDateDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview
```

```
Class: R_integerDomain
  EquivalentTo:
    LeaveRequest_calculatePeriodLength_integer or
    PerformanceReview_calculatePeriodLength_integer

DataProperty: r_integer
  Domain: R_integerDomain
  Range: integer

ObjectProperty: calculatePeriodLength_inv
  InverseOf: calculatePeriodLength
ObjectProperty: calculatePeriodLength
  Domain: CalculatePeriodLengthDomain
  Range: CalculatePeriodLengthRange

Class: LeaveRequest_calculatePeriodLength_integer
  SubClassOf:
    calculatePeriodLength_inv exactly 1 Thing,
    r_integer exactly 1 integer
  HasKey:
    calculatePeriodLength_inv

Class: PerformanceReview_calculatePeriodLength_integer
  SubClassOf:
    calculatePeriodLength_inv exactly 1 Thing,
    r_integer exactly 1 integer
  HasKey:
    calculatePeriodLength_inv

DisjointClasses: Employee, LeaveRequest, LeaveRequestStatus,
  PerformanceReview
```

E.2 Translation of Redesigned LeaveRequest and PerformanceReview Classes

Here we provide the complete translation of the UML class diagram of Figure 2.24 to OWL 2 Manchester Syntax [57].

Class: PerformanceReview
SubClassOf:
 employee exactly 1 Thing,
 period exactly 1 Thing,
 manager exactly 1 Thing
HasKey:
 employee, period

Class: LeaveRequest
SubClassOf:
 id exactly 1 integer,
 reason exactly 1 string,
 period exactly 1 Thing,
 status exactly 1 Thing
HasKey:
 id

ObjectProperty: employee
 Domain: PerformanceReview
 Range: Employee
ObjectProperty: manager
 Domain: PerformanceReview
 Range: Employee

ObjectProperty: period
 Domain: PeriodDomain
 Range: Period

DataProperty: id
 Domain: LeaveRequest
 Range: integer
DataProperty: reason
 Domain: LeaveRequest
 Range: string
ObjectProperty: status
 Domain: LeaveRequest
 Range: LeaveRequestStatus

```
Class: Employee
Class: LeaveRequestStatus
Class: Period
  SubClassOf:
    fromDate exactly 1 string,
    toDate exactly 1 string,
    calculatePeriodLength
    some Period_calculatePeriodLength_integer

Class: PeriodDomain
  EquivalentTo:
    LeaveRequest or
    PerformanceReview

DataProperty: r_integer
  Domain: Period_calculatePeriodLength_integer
  Range: integer

ObjectProperty: calculatePeriodLength_inv
  InverseOf: calculatePeriodLength
ObjectProperty: calculatePeriodLength
  Domain: Period
  Range: Period_calculatePeriodLength_integer

Class: Period_calculatePeriodLength_integer
  SubClassOf:
    calculatePeriodLength_inv exactly 1 Thing,
    r_integer exactly 1 integer
  HasKey:
    calculatePeriodLength_inv

DataProperty: fromDate
  Domain: Period
  Range: string
DataProperty: toDate
  Domain: Period
  Range: string

DisjointClasses: Employee, LeaveRequest, LeaveRequestStatus
PerformanceReview, Period
```

Appendix F

Low Inheritance Cohesion Examples Translated to OWL 2

F.1 Translation of Rectangle and Square Classes with Low Inheritance Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.25 to OWL 2 Manchester Syntax [57].

```
DataProperty: width
  Domain: Rectangle
  Range: integer
DataProperty: height
  Domain: Rectangle
  Range: integer

Class: Quadrilateral
Class: Rectangle
  SubClassOf:
    Quadrilateral
    width exactly 1 integer,
    height exactly 1 integer
Class: Square
  SubClassOf: Rectangle
```

F.2 Translation of Redesigned Rectangle and Square Classes

Here we provide the complete translation of the UML class diagram of Figure 2.26 to OWL 2 Manchester Syntax [57].

```
DataProperty: width
  Domain: Rectangle
  Range: integer
DataProperty: height
  Domain: Rectangle
  Range: integer
DataProperty: length
  Domain: Square
  Range: integer

Class: Quadrilateral
Class: Square
  SubClassOf: Quadrilateral,
    length exactly 1 integer
Class: Rectangle
  SubClassOf: Quadrilateral,
    width exactly 1 integer
    height exactly 1 integer
```

F.3 Translation of Bird Inheritance Hierarchy with Low Inheritance Cohesion

Here we provide the complete translation of the UML class diagram of Figure 2.27 to OWL 2 Manchester Syntax [57].

```
ObjectProperty: fly_inv
  InverseOf: fly
ObjectProperty: fly
  Domain: Bird
  Range: Bird_fly

ObjectProperty: walk_inv
  InverseOf: walk
ObjectProperty: walk
  Domain: Bird
  Range: Bird_walk
```

```
Class: Bird
  SubClassOf:
    fly some Bird_fly
    walk some Bird_walk
Class: Eagle
  SubClassOf: Bird
Class: Penguin
  SubClassOf: Bird

Class: Bird_fly
  SubClassOf: fly_inv exactly 1 Thing
Class: Bird_walk
  SubClassOf: walk_inv exactly 1 Thing
```

F.4 Translation of Redesigned Bird Inheritance Hierarchy

Here we provide the complete translation of the UML class diagram of Figure 2.28 to OWL 2 Manchester Syntax [57].

```
ObjectProperty: fly_inv
  InverseOf: fly
ObjectProperty: fly
  Domain: FlyingBird
  Range: FlyingBird_fly

ObjectProperty: walk_inv
  InverseOf: walk
ObjectProperty: walk
  Domain: Bird
  Range: Bird_walk

Class: FlyingBird_fly
  SubClassOf: fly_inv exactly 1 Thing
Class: Bird_walk
  SubClassOf: walk_inv exactly 1 Thing
```

```
Class: Bird
  SubClassOf:
    walk some Bird_walk
Class: FlyingBird
  SubClassOf: Bird
    fly some FlyingBird_fly
Class: FlightlessBird
  SubClassOf: Bird
Class: Eagle
  SubClassOf: FlyingBird
Class: Penguin
  SubClassOf: FlightlessBird
```

Bibliography

- [1] protégé. url=<http://protege.stanford.edu/>.
- [2] TopBraid Composer: Maestro Edition. url=<http://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/>.
- [3] OMG Object Constraint Language (Version 2.4). Technical Report formal/2014-02-01, Object Management Group, February 2014.
- [4] A. Artale, D. Calvanese, and A. Ibáñez-García. Full satisfiability of UML class diagrams. In J. Parsons, M. Saeki, P. Shoval, C. Woo, and Y. Wand, editors, *Conceptual Modeling – ER 2010*, volume 6412 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin Heidelberg, 2010.
- [5] A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, and M. Zakharyashev. Reasoning over Extended ER Models. In C. Parent, K. Schewe, V. C. Storey, and B. Thalheim, editors, *Conceptual Modeling - ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 277–292. Springer Berlin Heidelberg, 2007.
- [6] F. Baader. What’s new in Description Logics. *Informatik-Spektrum*, 34(5):434–442, 2011.
- [7] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conferences on Artificial Intelligence*, pages 452–457. Morgan Kaufmann Publishers Inc., 1991.
- [8] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The description logic handbook: theory, implementation and applications*, pages 43–95. Cambridge University Press, New York, USA, 2003.
- [9] F. Barbier, B. Henderson-Sellers, A. L. Parc, and J. Bruel. Formalization of the whole-part relationship in the Unified Modeling Language. *IEEE Transactions on Software Engineering*, 29(5):459–470, 2003.

-
- [10] D. Berardi. Using DLs to reason on UML class diagrams. In G. Görz, V. Haarslev, C. Lutz, and R. Möller, editors, *Proceedings of the KI-2002 Workshop on Applications of Description Logics*, volume 63. CEUR Worksop Proceedings, September 2002.
- [11] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams using description logic based systems. In G. Görz, V. Haarslev, C. Lutz, and R. Möller, editors, *Proceedings of the KI-2001 Workshop on Applications of Description Logics*, volume 44. CEUR Worksop Proceedings, 2001.
- [12] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams is EXPTIME-hard. In D. Calvanese, G. D. Giacomo, and E. Franconi, editors, *Proceedings of the 16th International Workshop on Description Logics*, volume 81. CEUR Workshop Proceedings, 2003.
- [13] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, Oct. 2005.
- [14] D. Bildhauer. On the relationships between subsetting, redefinition and association specialization. In D. Costal, C. Gómez, and G. Guizzardi, editors, *Databases and Information Systems: Proceedings of the Ninth International Baltic Conference*, Riga, Latvia, 2010.
- [15] T. Bittner and M. Donnelly. Computational ontologies of parthood, componenthood, and containment. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 382–387. Professional Book Center, 2005.
- [16] B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, January 2001.
- [17] B. W. Boehm. Guidelines for verifying and validating software requirements and design specifications. In P. A. Samet, editor, *Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing*, pages 711–719. North-Holland Publishing Company, September 1979.
- [18] G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. *Object-oriented analysis and design with applications*. Addison-Wesley Professional, 3rd edition, April 2007.
- [19] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.

-
- [20] B. Braga, J. Almeida, G. Guizzardi, and A. Benevides. Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. 6(1-2):55–63, March 2010.
- [21] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.
- [22] F. P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [23] J. Cabot, R. Clariso, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, July 2014.
- [24] A. Calí, D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning on UML class diagrams in description logics. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD)*, volume 2083 of *Lecture Notes in Artificial Intelligence*. Springer, 2001.
- [25] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In M. Hacid, Z. W. Raś, D. A. Zighed, and Y. Kodratoff, editors, *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer Berlin Heidelberg, 2002.
- [26] D. Calvanese and G. D. Giacomo. An introduction to description logics. In F. Baader, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.
- [27] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, and R. Rosati. Ontologies and databases: The DL-Lite approach. In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset, and R. A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer Berlin Heidelberg, 2009.
- [28] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. DL-Lite: Tractable Description Logics for Ontologies. In A. Cohn, editor, *Proceedings of the 20th National Conference on Artificial Intelligence*, volume 2 of *Assosiation for the Advancement of Artificial Intelligence*, pages 602–607. AAAI Press, 2005.
- [29] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information*

- Systems*, volume 436 of *The Springer International Series in Engineering and Computer Science*, pages 229–263. Springer US, 1998.
- [30] A. Cockburn. *Writing effective use cases*. Addison-Wesley Professional, 2000.
- [31] D. Costal, C. Gómez, and G. Guizzardi. Formal semantics and ontological analysis for understanding subsetting, specialization and redefinition of associations in UML. In M. A. Jeusfeld, L. M. L. Delcambre, and T. Ling, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin Heidelberg, 2011.
- [32] C. J. Date. *Relational database: selected writings*. Addison Wesley Publishing Company, 1986.
- [33] C. J. Date. *The relational database dictionary*. FirstPress. Apress, 2008.
- [34] C. Denger and T. Olsson. *Quality assurance in requirements engineering*. Springer Berlin Heidelberg, 2005.
- [35] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web Journal*, 2(2):71–87, 2011.
- [36] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. Center for the Study of Language and Information, 1996.
- [37] H. Du and L. Wery. Micro: a normalization tool for relational database designers. *Journal of Network and Computer Applications*, 22(4):215–232, 1999.
- [38] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt.
- [39] L. H. Etzkorn and H. S. Delugach. Towards a semantic metrics suite for object-oriented design. In Q. Li, D. Firesmith, R. Riehle, and B. Meyer, editors, *Proceedings of the 34th Technology of Object-Oriented Languages and Systems*, pages 71–80. IEEE, 2000.
- [40] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2003.
- [41] R. Falbo, G. Guizzardi, A. Gangemi, and V. Presutti. Ontology patterns: clarifying concepts and terminology. In A. Gangemi, M. Gruninger, K. Hammar, L. Lefort, V. Presutti, and A. Scherp, editors, *4th International Workshop on Ontologies and Semantic Patterns*, volume 1188. CEUR Worksop Proceedings, 2013.

- [42] M. Fowler. *Analysis patterns - reusable object models*. Addison-Wesley Professional, 1997.
- [43] M. Fowler. *UML distilled: a brief guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 3rd edition, 2003.
- [44] G. D. Giacomo. Description logics for conceptual data modeling in UML. "http://www.eecs.yorku.ca/course_archive/2010-11/F/6390A/DLmaterial/DeGiacomo-2-uml-dls2up.pdf".
- [45] C. Golbreich, M. Horridge, I. Horrocks, B. Motik, and R. Shearer. OBO and OWL: Leveraging semantic web technologies for the life sciences. In K. Aberer, P. Cudré-Mauroux, K. Choi, N. Noy, D. Allemang, K. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, and G. Schreiber, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 169–182. Springer-Verlag Heidelberg Berlin, 2007.
- [46] H. Gomma. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, March 2011.
- [47] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. OWL 2: the next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, November 2008.
- [48] G. Guizzardi. *Ontological foundations for structural conceptual models*. PhD thesis, 2005.
- [49] T. A. Halpin and H. A. Proper. Subtyping and polymorphism in object-role modelling. *Data & Knowledge Engineering*, 15(3):251–281, 1995.
- [50] H. Harmse, A. Britz, and A. Gerber. Scenario testing on UML class diagrams using description logics. "<http://www.cair.za.net/research/outputs/scenario-testing-uml-class-diagrams-using-description-logic>", 2013.
- [51] H. Harmse, K. Britz, A. Gerber, and D. Moodley. Scenario testing using formal ontologies. In G. Guizzardi, O. Pastor, Y. Wand, S. D. Cesare, F. Gailly, M. Lycett, and C. Partridge, editors, *1st Joint Workshop ONTO.COM / ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering*, volume 1301 of *CEUR Workshop Proceedings*. CEUR Workshop Proceedings, 2014.
- [52] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. In A. Paepcke, editor, *Conference proceedings on Object-oriented programming systems, languages, and applications*, volume 26 of *ACM Special Interest Group on Programming Languages Notices*, pages 171–183. ACM, November 1991.

- [53] B. Henderson-Sellers and F. Barbier. What is this thing called aggregation? In R. Mitchell, A. C. Wills, J. Bosch, and B. Meyer, editors, *Proceedings of 29th Technology of Object-Oriented Languages and Systems*, pages 236–250. IEEE, 1999.
- [54] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 web ontology language primer. "[http=http://www.w3.org/TR/owl2-primer/](http://www.w3.org/TR/owl2-primer/)".
- [55] S. Hong. A class normalization approach to the design of object-oriented databases. In *Proceedings of the 5th Technology of Object-Oriented Languages and Systems*. Prentice Hall, 1991.
- [56] M. Horridge. *Justification based explanation in ontologies*. PhD thesis, University of Manchester, 2011.
- [57] M. Horridge and P. F. Patel-Schneider. OWL 2 web ontology language Manchester syntax. url=<http://www.w3.org/TR/owl2-manchester-syntax/>, December 2012.
- [58] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible *SR_{OIQ}*. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 57–67. AAAI Press, 2006.
- [59] I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):345–357, October 2004.
- [60] I. Horrocks and U. Sattler. Ontology reasoning in the *SHOQ^(D)* description logic. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, volume 1, pages 199–204. Morgan Kaufmann Publishers Inc., 2001.
- [61] I. Horrocks and U. Sattler. A tableau decision procedure for *SHOIQ*. *Journal of Automated Reasoning*, 39(3):249–276, 2007.
- [62] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic *SHIQ*. In D. A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 482–496. Springer-Verlag London, 2000.
- [63] ISO. *Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure*, iso/iec 19505-1 edition, April 2012.
- [64] ISO. *Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure*, iso/iec 19505-2 edition, April 2012.

- [65] D. Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, 2006.
- [66] A. Kalyanpur. *Debugging and repair of owl ontologies*. PhD thesis, Maryland, USA, 2006.
- [67] C. Kaner. An Introduction to Scenario Testing. Technical report, 2003.
- [68] Y. Kazakov. *RIQ and SROIQ Are Harder than SHOIQ*. In G. Brewka and J. Lang, editors, *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, pages 274–284. AAAI Press, 2008.
- [69] C. Keet. Introduction to part-whole relations: mereology, conceptual modelling and mathematical aspects. Technical report, Free University of Bozen-Bolzano, Oct. 2006.
- [70] C. M. Keet. A formal comparison of conceptual data modeling languages. In T. Halpin, E. Proper, J. Krogstie, X. Franch, E. Hunt, and R. Coletta, editors, *Proceedings of the 13th International Workshop on Exploring Modeling Methods in Systems Analysis and Design*, volume 337, pages 25–39. CEUR Worksop Proceedings, 2008.
- [71] C. M. Keet. Detecting and revising flaws in OWL object property expressions. In A. ten Teije, J. Völker, S. Handschuh, H. Stuckenschmidt, M. d’Aquin, A. Nikolov, N. Aussenac-Gilles, and N. Hernandez, editors, *Proceedings of the 18th International Conference on Knowledge Engineering and Knowledge Management*, volume 7603 of *Lecture Notes in Computer Science*, pages 252–266. Springer Berlin Heidelberg, 2012.
- [72] M. Khosrow-Pour. *Dictionary of information science and technology*. IGI Global, 2007.
- [73] S. Klarman. ABox abduction in description logic. Master’s thesis, Universiteit van Amsterdam, 2008.
- [74] S. Klarman, U. Endriss, and S. Schlobach. ABox abduction in the description logic *ALC*. *Journal of Automated Reasoning*, 46(1):43–80, 2011.
- [75] P. Kroha and J. Gayo. *Using semantic web technology in requirements specifications*. Chemnitzer Informatik-Berichte. Technische Universität Wien, 2008.
- [76] M. Krötzsch. *Description logic rules*. PhD thesis, Karlsruhe Institute of Technology, 2010.
- [77] M. Krötzsch, F. Simančík, and I. Horrocks. A description logic primer. *Computing Research Repository*, abs/1201.4089, 2012.
- [78] M. Krötzsch, F. Simančík, and I. Horrocks. Description logics. *IEEE Intelligent Systems*, 29(1):12–19, 2014.

-
- [79] H. Kung and L. Kung. An interactive tool to improve learning of data modeling: a survey study. *Journal of Computing Sciences in Colleges*, 28(4):11–18, Apr. 2013.
- [80] G. Lakoff. *Women, fire and dangerous things: what categories reveal about the mind*. University of Chicago Press, Chicago, 1990.
- [81] M. Lanza and R. Marinescu. *Object-Oriented metrics in practice*. Springer-Verlag Berlin Heidelberg, 2006.
- [82] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process*. Prentice Hall, 2nd edition, 2001.
- [83] B. Liskov. Data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications*, volume 23 of *ACM Special Interest Group on Programming Languages Notices*, pages 17–34, New York, USA, 3 1987. ACM.
- [84] C. Lutz, C. Areces, I. Horrocks, and U. Sattler. Keys, nominals, and concrete domains. *Journal of Artificial Intelligence Research*, 23:667–726, 2005.
- [85] C. Lutz, U. Sattler, and L. Tendera. The complexity of finite model reasoning in description logics. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, volume 199 of *Information and Computation*, pages 132–171. Elsevier, 2005.
- [86] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, March-April 2008.
- [87] R. Martin and M. Micah. *Agile principles, patterns, and practices in C#*. Prentice Hall, 2006.
- [88] R. C. Martin, M. C. Feathers, T. R. Ottinger, J. J. Langr, B. L. S. J. W. Grenning, and K. D. Wampler. *Clean code: a handbook of agile software craftsmanship*. Robert C. Martin Series. Prentice Hall, New Jersey, USA, 2008.
- [89] B. Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, March 1997.
- [90] B. Motik and I. Horrocks. OWL datatypes: design and implementation. In A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, and K. Thirunarayan, editors, *Proceedings of the 7th International Semantic Web Conference*, volume 5318 of *Lecture Notes Computer Science*, pages 307–322. Springer, 2008.

- [91] B. Motik, P. F. Patel-Schneider, and B. C. Grau. OWL 2 web ontology language: direct semantics. url=<http://www.w3.org/TR/owl2-direct-semantics/>, December 2012.
- [92] B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 web ontology language: structural specification and functional-style syntax. url=<http://www.w3.org/TR/owl2-syntax/>, 2012.
- [93] J. Mylopoulos. Conceptual modelling and Telos. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. John Wiley & Sons, 1992.
- [94] D. Nardi and R. Brachman. An introduction to description logics. In F. Baader, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*, pages 1–40. Cambridge University Press, New York, NY, 2003.
- [95] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In A. Finkelstein, editor, *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.
- [96] A. Olivé. *Conceptual modeling of information systems*. Springer, 2007.
- [97] B. Parsia, U. Sattler, and T. Schneider. Easy Keys for OWL. In C. Dolbear, A. Ruttenberg, and U. Sattler, editors, *Proceedings of the 5th Workshop on OWL: Experiences and Directions*, volume 432 of *CEUR Workshop Proceedings*. CEUR Workshop Proceedings, 2008.
- [98] B. Parsia, E. Sirin, and A. Kalyanpur. Debugging OWL ontologies. In A. Ellis and T. Hagino, editors, *Proceedings of the 14th international conference on World Wide Web*, Chiba, Japan, May 2005. ACM.
- [99] Protege. Protege 4.x Anonymous Classes. url=<http://protegewiki.stanford.edu/wiki/P4AnonymousClasses>, 2009.
- [100] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, 73:1–22, 2012.
- [101] C. Rolland and C. Salinesi. Modeling goals and reasoning with them. In A. Aurum and C. Wohlin, editors, *Engineering and Managing Software Requirements*. Springer Berlin Heidelberg, 2005.
- [102] C. Roussey and O. Zamazal. Antipattern detection: how to debug an ontology without a reasoner. In P. Lambrix, G. Qi, M. Horridge, and B. Parsia, editors, *Proceedings of the*

- 2nd International Workshop on Debugging Ontologies and Ontology Mappings*, volume 999, pages 45–56. CEUR Workshop Proceedings, 2013.
- [103] S. Rudolph. Foundations of description logics. In A. Polleres, C. d’Amato, M. Arenas, S. Handschuh, P. Kroner, S. Ossowski, and P. F. Patel-Schneider, editors, *Reasoning Web. Semantic Technologies for the Web of Data 7th International Summer School*, volume 6848 of *Lecture Notes in Computer Science*, pages 76–136. Springer, 2011.
- [104] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual*. Addison Wesley, 2nd edition, 2005.
- [105] U. Sattler, D. Calvanese, and R. Molitor. Relationships with other formalisms. In F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The description logic handbook: theory, implementation and applications*, pages 137–177. Cambridge University Press, 2003.
- [106] S. Schlobach and R. Cornet. Non-Standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 2003.
- [107] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial intelligence*, 48(1):1–26, 1991.
- [108] L. Schröder and D. Pattinson. How many toes do I have? Parthood and number restrictions in description logics. In G. Brewka and J. Lang, editors, *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 307–218. AAAI Press, 2008.
- [109] P. Shoval. *Functional and object oriented analysis and design: an integrated methodology*. IGI Global, 2006.
- [110] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1341–1344. IEEE Computer Society, 2010.
- [111] V. C. Storey, C. B. Thompson, and S. Ram. Understanding database design expertise. *Data & Knowledge Engineering*, 16(2):97–124, 1995.
- [112] M. Szlenk. Formal semantics and reasoning about UML class diagram. In W. Zamojski, J. Mazurkiewicz, J. Sugier, and T. Walkowiak, editors, *Proceedings of the International Conference on Dependability of Computer Systems*, pages 51–59. IEEE, 2006.
- [113] A. ter Hofstede, H. Proper, and T. van der Weide. A conceptual language for the description and manipulation of complex information models. In G. Gupta, editor, *17th*

- Annual Computer Science Conference*, volume 16, pages 157–167. Australian Computer Science Communications, January 1994.
- [114] A. Tort and A. Olivé. An approach to testing conceptual schemas. *Data & Knowledge Engineering*, 69(6):598–618, 2010.
- [115] Y. Wand, D. E. Monarchi, J. Parsons, and C. C. Woo. Theoretical foundations for conceptual modelling in information systems development. *Decision Support Systems*, 15(4):285–304, 1995.
- [116] R. S. Wazlawick. *Object-oriented analysis and design for information systems: modeling with UML, OCL and IFML*. Morgan Kaufmann, 2014.
- [117] P. Wegner. Classification in object-oriented systems. In P. Wegner and B. Shriver, editors, *Proceedings of the Special Interest Group on Programming Languages Workshop on Object-oriented Programming*, volume 21, pages 173–182. ACM, 1986.
- [118] J. Zedlitz, J. Jörke, and N. Luttenberger. From UML to OWL 2. In D. Lukose, A. Ahmad, and A. Suliman, editors, *Third Knowledge Technology Week*, volume 295 of *Communications in Computer and Information Science*, pages 154–163. Springer Berlin Heidelberg, 2012.
- [119] J. Zedlitz and N. Luttenberger. Data types in UML and OWL 2. In A. Cheptsov, editor, *Proceedings of the 7th International Conference on Advances in Semantic Processing*. ThinkMind, 2013.