

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

TIAGO PRINCE SALES

ONTOLOGY VALIDATION FOR MANAGERS

VITÓRIA, BRASIL

2014

TIAGO PRINCE SALES

ONTOLOGY VALIDATION FOR MANAGERS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Informática.

Orientador: Prof. Dr. Giancarlo Guizzardi

VITÓRIA, BRASIL

2014

TIAGO PRINCE SALES

ONTOLOGY VALIDATION FOR MANAGERS

Dissertação submetida ao programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Informática.

Aprovada em 10 de Outubro de 2014.

COMISSÃO EXAMINADORA

Prof. Dr. Giancarlo Guizzardi - Orientador
Universidade Federal do Espírito Santo

Prof. Dr. João Paulo Andrade Almeida
Universidade Federal do Espírito Santo

Ig Ibert Bittencourt Santana Pinto
Universidade Federal do Alagoas

“The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.”

- Edsger W. Dijkstra

ACKNOWLEDGEMENTS

I would like to thank my loving wife, Camila. Your unwavering confidence in me has always encouraged me to shoot for the stars. Without your support, I would not have made it. I love you pumpkin!

I would like to thank my older brother, Lucas. My eternal bro! You are my best friend, period.

I would also thank my parents, Joseph and Magali. Everything I am and will ever become is because of you. Even though life may take me far away, you will always mean the world to me.

I would like to give special thanks to my dear friend Ernani. During this whole research, I could always count on you. Thank you for the discussions, for remembering me to be practical, and most importantly, for all your insightful comments on the previous versions of this thesis.

I would also like to show my appreciation to my friend John. It is awesome to do research with you! I know that the tools develop in this thesis would not be nearly as good as they are today without you. One more thing... OLED rules!

I would also like to thank Prof. Giancarlo Guizzardi, for giving me the opportunity to work with you. Thank you for all the valuable counsels and guidance throughout the last years. I learned a lot from you, and I hope to keep learning.

I would like to thank all my colleagues and professors at NEMO. It has been an amazing experience to be a part of this research group. Particular thanks to Prof. João Paulo Almeida, Prof. Renata Guizzardi and Prof. Anilton Garcia, for all the valuable lessons; and to my friends Pedro Paulo, Freddy, Bernardo, Júlio, Carlos, Sobral, Victório, Maria das Graças and Bassetti for sharing this academic journey with me.

Lastly, I thank my friends that are not on the academy. Life without you guys would not be nearly as fun! Tomas, Viola, Carol, Hugo e Faixinha.

ABSTRACT

Ontology-driven conceptual modeling is the activity of capturing and formalizing how a community perceives a domain of interest, using modeling primitives inherited from a foundational ontology. OntoUML is an example of a language that supports such activity, whose design derives from the Unified Foundational Ontology (UFO).

Ontologies, in the sense of reference conceptual models, are useful in many fields. They include model-driven development of software systems, development of knowledge-based application (in the context of Semantic Web), semantic interoperability between information systems, and evaluation of modeling languages, to cite some. Regardless of the application, the quality of an ontology is directly related to the quality of the results.

Ontology and conceptual model quality encompasses a vast range of criteria. The validation activity aims to improve the domain appropriateness of a model. This means to help improve modeler's confidence in saying: "I built the right model for my domain".

This thesis presents a validation framework usable by "managers" of the ontology world, i.e. modelers that are not experts in validation, logics and formal methods. The framework contains techniques and tools to help modelers systematically improve the quality of their models without demanding costly learning requirements. We build our framework on two conceptual pillars: model simulation and anti-patterns.

RESUMO

Modelagem conceitual orientada por ontologias é atividade de capturar e formalizar a forma que uma comunidade entende e classifica um domínio de interesse, usando para isso primitivas de modelagem herdadas de uma ontologia de fundamentação. OntoUML é um exemplo de linguagem para esse tipo de atividade, cujo meta-modelo é definido com base na Unified Foundational Ontology (UFO).

Ontologias, enquanto modelos conceituais de referência, são utilizadas para diversos fins. Dentre eles, destacam-se o desenvolvimento de sistemas de informação por abordagens orientadas a modelos, o desenvolvimento de aplicações para web-semântica (knowledge-based applications), a interoperabilidade semântica entre sistemas e a avaliação de linguagens de modelagem. Independente do propósito para o qual está sendo desenvolvida, a qualidade da ontologia está diretamente relacionada a qualidade dos resultados de sua aplicação.

Qualidade de ontologias e de modelos conceituais em geral, abrangem um vasto leque de critérios. A atividade de validação, no entanto, tem como objetivo aumentar um subconjunto desses critérios apenas, para que modeladores tenham confiança de que eles capturaram corretamente a conceituação a cerca de um domínio em suas ontologias.

Essa dissertação propõe um *framework* de validação utilizável por “gerentes” do mundo das ontologias, isto é, modeladores que conhecem a linguagem de modelagem mas não tem conhecimentos profundos de validação, lógica e métodos formais. Esse framework contém técnicas e ferramentas para sistematicamente ajudar modeladores a melhorarem a qualidade de seus modelos, sem que para isso requeira dispendiosos estudos prévios. O framework de validação é construído em cima de dois pilares: simulação de modelos e anti-padrões.

LIST OF FIGURES

Figure 1. Basic UFO definitions.....	27
Figure 2. Universals, individuals and their relations in UFO.	28
Figure 3. UFO's hierarchy of substantials.	32
Figure 4. UFO's hierarchy of moments.....	34
Figure 5. UFO's hierarchy of relation universals.....	36
Figure 6. Intended and possible model instantiations, adapted from (GANGEMI et al., 2005).....	40
Figure 7. Partial OntoUML diagram with identity issues in all classes.....	70
Figure 8. Example of a linear branch.....	80
Figure 9. Alternative futures example.....	81
Figure 10. Counterfactual world example.....	82
Figure 11. Internship conceptual model.....	84
Figure 12. World with population of four individuals.	84
Figure 13. Population Variability: constant objects scenario - Current world.....	86
Figure 14. Population Variability: constant objects scenario - Future world.	86
Figure 15. Population Growth (Incremental) scenario - Current World.....	88
Figure 16. Population Growth (Incremental) scenario - Future World.	88
Figure 17. Extension Comparison scenario example.	93
Figure 18. Association Changeability (Generic Dependency) - (a) Current World and (b) Future World	99
Figure 19. Cardinality Value scenario (at least three relations).	101
Figure 20. Partial OntoUML model of the domain of criminal investigation.	107
Figure 21. Fragment of the O3 ontology that contains an occurrence of the AssCyc anti-pattern.	116
Figure 22. A possible instantiation of the O3 ontology, exemplifying an open instance cycle.	117
Figure 23. A possible instantiation of the O3 ontology, exemplifying a closed instance cycle.	117
Figure 24. Fragment of the MGIC that exemplifies BinOver.	123

Figure 25. On the left, in white, a possible world in which the “isConnectedTo” relation is transitive and acyclic; on the right, in grey, a world where “isConnectedTo” is reflexive, symmetric and cyclic	123
Figure 26. Simplified fragment of the FIFA Football Model characterizing a Declnt occurrence.....	126
Figure 27. DepPhase occurrence identified in the Quality Assurance Model.	129
Figure 28. Application of the four role specialization patterns.	132
Figure 29. Translated and simplified fragment of the MPOG Ontology Draft exemplifying the GSRig anti-pattern.....	136
Figure 30. Simplified fragment of the IDAF model that depicts the HetColl anti-pattern.	142
Figure 31. <i>HomoFunc</i> occurrence encountered in the PAS 77:2006 ontology.	145
Figure 32. ImpAbs occurrence identified in the ECG Ontology.	149
Figure 33. Simplified fragment of the MGIC that characterizes a MixIden occurrence.	152
Figure 34. Simplified MixRig occurrence identified in the MGIC ontology	155
Figure 35. MultDep example extracted from the OntoEmerge ontology.	160
Figure 36. PartOver occurrence identified in the MGIC ontology.	161
Figure 37. Fragment of the CSHG ontology characterizing a RelComp.	163
Figure 38. Generated examples of the CSHG excerpt. On the left, an expected instantiation. On the right, an undesired one.	164
Figure 39. RelOver occurrence encountered in the UFO-S ontology.	171
Figure 40. Overlapping mediated types without exclusiveness constraint.	171
Figure 41. Simultaneous role instantiation with exclusive relators.....	172
Figure 42. Simplified fragment of the ECG ontology characterizing a RelRig occurrence.....	176
Figure 43. RelSpec occurrence identified in the OntoBio ontology.....	181
Figure 44. Characterization of the redefinition constraint.	181
Figure 45. Characterization of the subsetting constraint	182
Figure 46. Characterization of the disjointness constraint.	182
Figure 47. RepRel occurrence identified in the CMTO ontology.....	186
Figure 48. Possible world generated for the diagram in Listing 18 without adding the uniqueness constraint regarding the relator <i>Change Request</i>	187
Figure 49. UndefFormal occurrence identified in the OVO ontology.	191

Figure 50. UndefPhase occurrences identified in a) the MPOG Ontology Draft and b) the Health Organization Model	194
Figure 51. WholeOver occurrence identified in the IT Infrastructure model.....	195
Figure 52. An ECG ontology's simplified excerpt illustrating PAR.	199
Figure 53. Detailing of the protocol adopted for anti-pattern identification through visual simulation	209
Figure 54. Distribution of the class' stereotypes within the MGIC Ontology.	227
Figure 55. Distribution of association's stereotypes within the MGIC Ontology.....	227
Figure 56. Simplified and translated excerpt of the MGIC ontology which about the regulation domain.....	232
Figure 57. OLEDv0.9.34 screenshot running on Windows 8.1.....	238
Figure 58. Alloy meta-model fragment: module composition.....	240
Figure 59. Alloy meta-model fragment: paragraph composition.	241
Figure 60. Alloy meta-model fragment: command paragraph.....	242
Figure 61. Alloy meta-model fragment: expressions.	243
Figure 62. Simulation component within OLED.....	244
Figure 63. Custom theme: yellow boxes for objects, red ellipses for properties and grey hexagons for datatypes.	246
Figure 64. Anti-pattern identification dialog on OLED.....	247
Figure 65. Anti-pattern result dialog in OLED.....	248
Figure 66. RelRig's initial anti-pattern wizard page.	249
Figure 67. RelRig's wizard page example	250
Figure 68. RelRig's finishing page.....	251
Figure 69. Class' stereotype distribution in the G.805 Ontology.....	287
Figure 70. Association's stereotype distribution in the G. 805 Ontology.....	288
Figure 71. Distribution of OntoEmerge class' stereotypes.....	289
Figure 72. OntoEmerge's association stereotype distribution.....	290
Figure 73. Distribution of the class' stereotypes within the OntoBio ontology.	292
Figure 74. Association's stereotypes distribution in the OntoBio Ontology.....	292
Figure 75. Class distribution in the O3 Ontology.	293
Figure 76. Association's stereotype distribution in the O3 Ontology.....	294

LIST OF TABLES

Table 1. Class mapping to Alloy	53
Table 2. Mapping of OntoUML Associations in Alloy.....	55
Table 3. Mapping of attributes and association ends into Alloy	57
Table 4. Mapping of meronymic constraints and meta-properties.....	60
Table 5. Mapping of relators, mediations, modes, qualities and characterizations....	62
Table 6. Mapping of datatypes and related elements and constraints in Alloy	64
Table 7. Mapping of String, Boolean and Integer.	66
Table 8. Mapping of generalizations and generalization sets.....	67
Table 9. Linear Branch – scenario description.	80
Table 10. Alternative Futures – scenario description.....	81
Table 11. Counterfactual Worlds – scenario description.	82
Table 12. Branch Depth – scenario description.....	83
Table 13. Population Size – scenario description.....	85
Table 14. Population Variability – scenario description.....	87
Table 15. Population Growth – scenario description.	89
Table 16. Extension Size – scenario description.....	90
Table 17. Temporal Extension Size – scenario description.....	91
Table 18. Extension Variability – scenario description.	92
Table 19. Extension Comparison – scenario description.....	93
Table 20. Extension Size Comparison – scenario description.....	94
Table 21. Multiple Instantiation – scenario description.	95
Table 22. Temporal Multiple Instantiation – scenario description.....	95
Table 23. Exclusive Instantiation – scenario description.	96
Table 24. Mandatory Anti-Rigidity – scenario description.....	97
Table 25. Pseudo-Rigid – scenario description.	98
Table 26. Association Changeability – scenario description.....	100
Table 27. Cardinality Value – scenario description.....	101
Table 28. Association Depth – scenario description.....	102
Table 29. Characterization of the AssCyc anti-pattern.	115
Table 30. Relevant binary properties for conceptual modeling.....	120
Table 31. Binary property values embedded in OntoUML's associations.	121

Table 32. Characterization of the <i>BinOver</i> anti-pattern.....	121
Table 33. Characterization of the <i>DeclInt</i> anti-pattern.....	125
Table 34. Characterization of the <i>DepPhase</i> anti-pattern.....	128
Table 35. Characterization of the <i>FreeRole</i> anti-pattern.....	133
Table 36. Characterization of the <i>GSRig</i> anti-pattern.....	137
Table 37. Characterization of the <i>HetColl</i> anti-pattern.....	140
Table 38. Characterization summary of the <i>HomoFunc</i> anti-pattern.....	144
Table 39. Characterization summary of the <i>ImpAbs</i> anti-pattern.....	147
Table 40. Characterization summary of the <i>MixIden</i> anti-pattern.....	150
Table 41. Characterization summary of the <i>MixRig</i> anti-pattern.....	153
Table 42. Characterization summary of the <i>MultDep</i> anti-pattern.....	157
Table 43. Characterization summary of <i>PartOver</i> the anti-pattern.....	161
Table 44. Characterization summary of the <i>RelComp</i> anti-pattern.....	166
Table 45. Characterization summary of the <i>RelOver</i> anti-pattern.....	169
Table 46. Characterization summary of the <i>Re/Rig</i> anti-pattern.....	174
Table 47. Characterization summary of the <i>RelSpec</i> anti-pattern.....	178
Table 48. Characterization summary of the <i>RepRel</i> anti-pattern.....	185
Table 49. Characterization summary of the <i>UndefFormal</i> anti-pattern.....	189
Table 50. Characterization summary of <i>UndefPhase</i> the anti-pattern.....	192
Table 51. Characterization summary of the <i>WholeOver</i> anti-pattern.....	196
Table 52. Summary description of all models in the repository.....	205
Table 53. Structural description of all models in the repository	207
Table 54. A summary of the results in the first study	210
Table 55. Results of the second anti-pattern empirical study.....	212
Table 56. Summary with all identified occurrences in all models.....	217
Table 57. Anti-Pattern frequency on investigated models.....	219
Table 58. Anti-Pattern frequency on models with required elements.....	220
Table 59. Anti-pattern appearance rate regarding model elements.....	221
Table 60. Summary of anti-pattern accuracy results.....	229
Table 61. Summary of the evaluation results from both studies.....	235
Table 62. Summary of the refactoring choices for all anti-patterns.....	311

LIST OF LISTINGS

Listing 1. World Structure module in Alloy.....	47
Listing 2. Ontological Properties module in Alloy.....	48
Listing 3. Skeleton structure of a generated Alloy module.....	50
Listing 4. OCL invariant generated to enforce closed cycles in the O3 fragment.	118
Listing 5. OCL invariant enforcing derivation by intersection.....	127
Listing 6. OCL invariant generated to enforce exclusive wholes.	161
Listing 7. OCL invariant to enforce existential composition.	164
Listing 8. OCL invariant characterizing the Right Universal Composition.	164
Listing 9. OCL invariant characterizing the Left Universal Composition.	165
Listing 10. OCL invariant that characterizes Forbidden Composition.	165
Listing 11. OCL invariant to enforce Custom Existential Composition.....	165
Listing 12. OCL invariant to enforce exclusive mediated types	172
Listing 13. Subsetting constraint written in OCL.....	177
Listing 14. Redefinition constraint written in OCL.....	177
Listing 15. Disjointness constraint written in OCL.....	178
Listing 16. OCL version of the Uniqueness Constraint for “current” relators.	184
Listing 17. Enforcing Uniqueness Constraint for “historical” relators using OCL.	185
Listing 18. OCL constraint to limit repeated relators in the CMTO ontology.	187
Listing 19. Exclusive parts constraint defined in OCL.....	196
Listing 20. Alloy’s module composition in EBNF.....	240
Listing 21. Alloy’s paragraph properties in EBNF.	241
Listing 22. Alloy’s command declaration in EBNF.	242
Listing 23. Alloy’s expression definition in EBNF.	242
Listing 24. An Alloy specification.	274

LIST OF ACRONYMS

ANTT – Agência Nacional de Transportes Terrestres

AssCyc – Association Cycle Anti-pattern

BinOver – Binary Relation between Overlapping Types Anti-pattern

CMTO – Configuration Management Task Ontology

DCFR – Domain Comparative Formal Relation

Declnt – Deceiving Intersection Anti-pattern

DepPhase – Relationally Dependent Phase Anti-pattern

EBNF – Extended Backus–Naur Form

ECG – Electrocardiogram

EMF – Eclipse Modeling Framework

ER – Entity Relationship

FreeRole – Free Role Specialization Anti-pattern

GS – Generalization Set

GSRig – Generalization Set with Mixed Rigidity

HetColl – Heterogeneous Collective Anti-pattern

HomoFunc – Homogeneous Functional Complex Anti-pattern

ImpAbs – Imprecise Abstraction Anti-pattern

MixIden – Mixin with Same Identity Anti-pattern

MixRig – Mixin with Same Rigidity Anti-pattern

MGIC – Modelo de Gestão da Informação e Conhecimento

MultDep – Multiple Relational Dependency Anti-pattern

ODP – Ontology Design Patterns

OMG – Object Management Group

ORM – Object-Role Modeling

OvO – Open proVenance Ontology

OWL – Ontology Web Language

PAR – Pseudo Anti Rigid

PartOver – Part Composed of Overlapping Wholes Anti-pattern

PDC – Pattern Diagram Class

RDF – Resource Description Framework

RefOntoUML – Reference OntoUML Metamodel

RelComp – Relation Composition Anti-pattern

RelOver – Relator Mediating Overlapping Types Anti-pattern

RelRig – Relator Mediating Rigid Types Anti-pattern

RelSpec – Relation Specialization Anti-pattern

RepRel – Repeatable Relator Instances Anti-pattern

UFO – Unified Foundational Ontology

UML – Unified Modeling Language

UndefFormal – Undefined Formal Association Anti-pattern

UndefPhase – Undefined Phase Partition Anti-pattern

VPML – Visual Pattern Modeling Language

WSMO – Web Service Modeling Ontology

WholeOver – Whole Composed by Overlapping Parts Anti-pattern

XML – *eXtension Markup Language*

SHORT TABLE OF CONTENTS

1	INTRODUCTION.....	17
2	THEORETICAL FOUNDATIONS.....	23
3	REVISITING ONTOUML2ALLOY	42
4	ANTI-PATTERNS IN ONTOLOGY-DRIVEN CONCEPTUAL MODELING.....	103
5	THE ANTI-PATTERN CATALOGUE	111
6	UNCOVERING SEMANTIC ANTI-PATTERNS.....	201
7	EVALUATING THE ANTI-PATTERN CATALOGUE	216
8	TOOL SUPPORT.....	236
9	CONCLUSIONS.....	252
	REFERENCES	263
	ANNEX A ALLOY	271
	ANNEX B AUXILIARY ALLOY MODULES	279
	APPENDIX A NOTEWORTHY CONCEPTUAL MODELS	286
	APPENDIX B ANTI-PATTERN ANALYSIS FLOWS.....	295
	APPENDIX C DETAILS OF THE MGIC STUDY	310

1 INTRODUCTION

1.1 CONTEXTUALIZATION

Ontology-driven conceptual modeling is the activity of capturing and formalizing how a community perceives a domain of interest, using modeling primitives inherited from a foundational ontology. This new branch of conceptual modeling improves traditional techniques by taking into consideration ontological properties, such as rigidity, identity and dependence (GUARINO; WELTY, 2009), all derived from a foundational ontology.

The reference ontology, product of such activity, serves a variety of purposes, which include:

- schema generation for knowledge bases (TBox), in the Semantic Web context (BARCELOS et al., 2013; ZAMBORLINI; GUIZZARDI, 2010);
- semantic interoperability between agents, systems and/or organizations (CALHAU; FALBO, 2010; GONÇALVES; GUIZZARDI; FILHO, 2011; NARDI; FALBO; ALMEIDA, 2013);
- model-driven software development, as means for generating code (PERGL; SALES; RYBOLA, 2013) and information models (CARRARETTO; ALMEIDA, 2012);
- standardization of a shared vocabulary for a community (in the sense of a reference model of consensus), as the service core ontology – UFO-S (NARDI et al., 2013) and the normative acts model (BARCELOS; GUIZZARDI; GARCIA, 2013);
- evaluation of modeling languages, as Azevedo et al (2011) perform for the ArchiMate enterprise architecture modeling language and Guizzardi et al. (2013) for the i* goal modeling language;

Recently, the interest in more expressive languages for conceptual modeling has increased, as it is evidenced by a request of the Object Management Group (OMG) for language proposals for the Semantic Information Model Federation (OMG, 2011a).

OntoUML is an example of such language. Its meta-model has been designed to comply with the ontological distinctions and axioms of a theoretically well-grounded foundational ontology, named the Unified Foundational Ontology (UFO) (GUIZZARDI, 2005).

OntoUML has been successfully employed in a number of industrial projects in several different domains, such as Oil and Gas (GUIZZARDI et al., 2010), Complex Digital Media Management (CAROLO; BURLAMAQUI, 2011), Telecommunications (BARCELOS et al., 2011), and Government (BASTOS et al., 2011; MPOG, 2011).

This thesis discusses the validation of OntoUML models. Particularly, we investigate techniques centered on the concepts of anti-pattern and model simulation. We aim to develop approaches that do not demand users to learn complex and time-consuming languages and methods. We strive to make ontology validation accessible for “Managers”. By managers, we do not mean the position commonly encountered in organizations, but the manager metaphor: someone who has a general knowledge about the technical processes, but is not an expert on them. We argue that this is the profile of a significant part of the ontology engineers. They are people with basic knowledge on the modeling language, on ontology engineering methods and validation tools. They are not, however experts in them, particularly in logics and formal methods, skills required for validation tasks.

1.2 MOTIVATION

The importance of formalizations that accurately capture their intended conceptualization has been recognized by both the traditional conceptual modeling (MOODY et al., 2003) and the semantic web (GANGEMI et al., 2006; SURE et al., 2004; VRANDEČIĆ, 2009) communities. As in the semantic web, the shared nature of ontology-driven conceptual models highlights the need of accurately defining them. Other ontologists are likely to reuse the ontologies we develop, possibly in ways that we cannot predict. Therefore, guaranteeing that we accurately formalized our ontologies is very important to guarantee a consistent reuse.

If ontologies are used in the context of model-driven software development, they play a key role in the process and directly affect the quality of the generated software products (MARÍN et al., 2010). Since model-driven processes generate software from a set of complimentary conceptual models, the models describe all the behavior, data and functionalities of a system. Thus, semantic errors in the model imply possible problems in the generated system.

Furthermore, in the cases where an ontology-driven conceptual model gives rise to semantic web applications, validation is also important. Vrandečić (2010) argues that, by improving ontology quality, reuse will increase, since others will gain confidence in the released ontologies. Besides, it facilitates integration and lower maintenance costs of ontologies.

If we use ontologies as reference models to achieve semantic interoperability, problems in their formalization may imply in data integration issues. An example of this type of problem is commonly known as the false agreement problem (GUARINO, 1998). An example of false agreement would be two classes stated as equivalent when in fact they formalize distinct concepts. For this reason, avoiding interoperability errors is particularly important on scenarios where data is valuable and the error tolerance is low.

This work is not only motivated by the importance of producing high-quality models but also by the recognition that it is complex to systematically guarantee such feature. Guizzardi (2010) makes a parallel between construction of large reference conceptual models and the programming of large computer systems, referencing the famous E. W. Dijkstra's ACM Turing lecture entitled "The Humble Programmer" (DIJKSTRA, 1972). Guizzardi's argument is that we, as "Humble Ontologists", must acknowledge the limitations of the human mind to address the large and fast increasingly intrinsic complexity of building ontologies. For this reason, conceptual modelers and ontologists should surround themselves with a number of suitable engineering tools to facilitate the tasks and improve the results of all activities in ontology engineering. As discussed in (GUIZZARDI, 2010), among these tools, we have modeling languages, e.g. OntoUML, and methodologies, e.g. NeOn (SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ; FERNÁNDEZ-LÓPEZ, 2012) and SABiO (FALBO; MENEZES; ROCHA, 1998), patterns and anti-patterns, as well as automated supporting environments for model

construction, verification and validation (BENEVIDES et al., 2010b; BRAGA et al., 2010).

Even though many recognize the importance of model quality and that guaranteeing it requires appropriate tool support, we argue that the wide adoption of such tools and techniques constantly faces a barrier: the learning curve. Many approaches require modelers to learn complex methods and languages. Furthermore, their validation capacity is directly dependent on their skills using these techniques. Although advanced knowledge on logics and formal methods are very useful for ontologists, we cannot expect every modeler to have such knowledge. To tackle this problem, we seek inspiration in (JANSSEN et al., 1999). In their work, entitled “Model Checking for Managers”, the authors propose an abstraction layer to allow untrained users to use model checking techniques on the validation of business process models. Our hypothesis is that we can do something analogous for ontology-driven conceptual modeling: develop easy to use tools and techniques that provide useful validation capabilities for OntoUML.

1.3 RESEARCH GOALS

The main goal of our research is to develop a validation framework for ontology-driven conceptual modeling. This framework should guide users throughout the validation process and aid them in systematically producing higher quality OntoUML models. Most importantly, it must not require any additional method or technique learning, such as processes or languages.

Our plan to achieve the main goal consists of the following specific goals:

- Revisit the simulation-based validation method proposed for OntoUML to reduce its learning requirements;
- Identify recurrent modeling decisions (anti-patterns) that are prone to lead to domain misrepresentation and develop an approach to use them to increase model quality;
- Develop adequate tool support for proposed approaches;
- Empirically assess the proposed techniques and tools.

1.4 THESIS STRUCTURE

We organized the remainder of this thesis as follows:

Chapter 2 (Theoretical Foundations) presents the ontological theory used throughout this thesis. Initially, an overview of the discipline of ontology-driven conceptual modeling is given. In the following, we present the most relevant ontological properties defined by UFO. Lastly, we briefly review OntoUML abstract and concrete syntax, presenting the fundamental concepts required to understand the remaining of this thesis.

Chapter 3 (Revisiting OntoUML2Alloy) concerns model simulation. We first present the motivation to revisit the transformation from OntoUML to Alloy, then the required notions of the Alloy language. In the sequence, we detail the new mappings of the transformation, followed by a comparison to the previous approaches. Lastly, we elaborate on the simulation scenarios that make the simulation tool more accessible for all users.

Chapters 4, 5, 6 and 7 are all anti-pattern related. Chapter 4 (Anti-patterns in Ontology-driven Conceptual Modeling) discusses our view on anti-patterns: definition, properties and inter-relations. We also elaborate on their role in ontology-driven conceptual modeling validation.

Chapter 5 (The Anti-Pattern Catalogue), presents the anti-pattern catalogue, defining their structure, context and refactoring solutions. For each anti-pattern type, we present an example encountered in a real model and the appropriate solution for it.

Chapter 6 (Uncovering Semantic Anti-Patterns) elaborates on the methods we used for uncovering anti-patterns: empirical analysis, foundational ontology evaluation and modeling language analysis.

Chapter 7 (Evaluating the Anti-Pattern Catalogue), presents and discusses the results of a study to evaluate the anti-pattern catalogue. We performed this analysis in the context of a project named “Modelo de Gestão da Informação e Conhecimento” (MGIC), conducted in cooperation with the Brazilian government.

Chapter 8 (Tool Support) discusses the implementation strategies and the tool we developed throughout this thesis to support our proposals.

Chapter 9 (Conclusions), discusses results, shortcomings, related works and questions that still demand further investigations.

Annex A (Alloy), presents an overview of the Alloy language, describing its semantics and syntax. We also review how to analyze structural models using Alloy.

Annex B (Auxiliary Alloy Modules) provides the complete source code of the auxiliary Alloy modules used in the transformation from OntoUML to Alloy.

Appendix A (Noteworthy Conceptual Models) discusses some distinguished OntoUML models we have in our repository, providing a more detailed description of the formalized domain, as well as discussions of their applications and further structural details.

Appendix B (Anti-Pattern Analysis Flows) presents the diagrams to analyze occurrences of all anti-patterns.

2 THEORETICAL FOUNDATIONS

2.1 ONTOLOGY-DRIVEN CONCEPTUAL MODELING

Mylopoulos (1992) defines conceptual modeling as the activity of formally describing relevant properties of a given domain for the purposes of understanding and communication. Expressivity is the main concern of such endeavor, which aims to accurately capturing domain knowledge and all its nuances. Unlike programs, which are supposed to be “read” by computers, we design conceptual models so other humans, not machines, can read and understand them.

Although employed for conceptual modeling, traditional techniques have not solely focused the main purpose of this activity: capturing domain knowledge. UML and ER are examples of such techniques. UML is designed to comply with an object-oriented view of software design, whilst ER (CHEN, 1976) to encounter a positive trade-off between conceptualization and implementation. The additional concerns of traditional approaches impair the expressivity of models produced by them, limiting its ability to capture domain knowledge.

In addition, traditional modeling languages do not clearly define their ontological commitments (GUARINO, 1998). This notion refers to the relation between the construct of a language and the real world concept it is supposed to represent. This commitment represents an agreement to use the shared vocabulary in a coherent and consistent manner in a given context.

The lack of expressivity and absence of explicit ontological commitment encountered in traditional methods, impair their capability to represent more precisely domain knowledge (GUIZZARDI, 2005, chap. 8). Models specified in such languages are prone to be ambiguous, incomplete and mix conceptual and implementation concerns. Note that we mean by ambiguity the real world meaning of language constructs and not its formal meaning.

The aforementioned language shortcomings, which affect the quality of conceptual models, emerge in the context of information systems, which is one of the main

applications of conceptual modeling. Low quality models influences the overall results of information system development. Moreover, the new reality of information technology, which requires systems to interoperate, further enhances the need of precisely defining meaning in conceptual models.

To overcome the aforementioned shortcomings, researchers argue that conceptual modeling and thus, conceptual modeling languages, should be driven by foundational ontologies (EVERMANN; WAND, 2005; GUIZZARDI, 2005; RECKER et al., 2011). These foundations are the most abstract, domain independent system of categories used by humans to describe and understand reality. The argumentation is built upon how ontology can better address important notions required in conceptual modeling, such as part-whole relations (GUIZZARDI, 2009) and specialization (COSTAL; GÓMEZ; GUIZZARDI, 2011) for structural conceptual modeling.

Guarino (1994) introduces the idea of the Ontological level in conceptual modeling. He differentiates it from the epistemological level by defining the concerns of these two levels. The ontological level is concerned with the nature of things, whilst the epistemological level's motivation is toward information demands. This view is also supported by Guizzardi (2010) for Ontology Engineering.

Proposed in (GUIZZARDI, 2005), the Unified Foundational Ontology (UFO) is an example of foundational ontology which is used to support an ontology-driven conceptual modeling language, named OntoUML. Cognitive Science, Linguistic, and Philosophy theories grounds UFO. Furthermore, the foundational ontology addresses important issues in structural conceptual modeling, such as identity and part-whole theory.

The ontology-driven conceptual modeling discipline thus, is akin to conceptual modeling. It is the activity of formally capturing domain knowledge driven by ontological foundations. Its main concern is to describe the nature of things that exist, their properties and relations, focusing on expressivity. The product of such activity is the ontology conceptual model (or reference conceptual model).

We take a moment here to highlight the duality of the ontology term. On one hand, we have these models that focus on accurately representing a domain of interest: the reference conceptual model. On the other hand, we have ontologies as the artificial

intelligence community defines them: reasoning artifacts that are the core of the semantic web. Notice that the main difference between them regards computational efficiency and expressivity. Ontology, as a reference conceptual model, is a very expressive description of reality, designed to be used and understood by humans. It does not meet, however, the demands for automated reasoning. Ontologies, as reasoning artifacts, are much less expressive on purpose. The reason is that the languages developed to build them were designed to be computationally efficient, and to allow automated reasoning, querying and so on. Resource Description Framework (RDF) (W3C, 2014) and Web Ontology Language (OWL) (W3C, 2012) are languages suitable to represent ontologies in the latter sense.

2.2 THE UNIFIED FOUNDATIONAL ONTOLOGY

In this section, we explain the theory defined in the foundational ontology alongside its reflection in the OntoUML syntax.

2.2.1 Individuals and Universals

The most fundamental notions in UFO are the concepts of individuals and universals, i.e. things and their types. Universals are space-time independent concepts that define patterns of features, like person, football match, sculpture, being heavier than, and so on. Individuals, conversely, are particular things that instantiate universals in space and time. You and me are individuals that instantiate the universal “Person”, as the 2014 World Cup Final between Germany and Argentina instantiates football match, the statue of David instantiates sculpture, and the relation that holds between a person that weighs 80 kg and another that weights 60kg instantiates “being heavier than”.

The concepts of universal, individual and the instantiation relation that holds between them are primitives of UFO’s theory. The theory assumes as universals all the things that can be instantiated and, as individuals, everything that instantiates universals. The instantiation relation only holds between individuals and universals, and imply that the individual has every characteristic assigned for the universal, e.g. if the universal

person defines features as height and weight, every individual must have a value for them.

To those familiar with traditional conceptual modeling techniques, universals are commonly referred to as classes, e.g. UML (OMG, 2011b) and OWL (W3C, 2012), entities, as in ER (CHEN, 1976), associations (as in UML), properties (as in OWL), and so on. Individuals, on the other hand, are usually known as objects, as in UML and ORM (HALPIN; MORGAN, 2008), instances, to cite a few.

The theory, though, does not exclude the possibility of something to behave simultaneously like an individual and a universal. Take for example, the token labeled as “Lion”. From one perspective, it is a universal, since it defines a group of features (e.g. being a carnivore, having four pawns, having fur) and has identifiable instances, like Simba and Mufasa (from the children’s cartoon Lion King). From another perspective, we can understand this token as an individual, which instantiates the “Species” universal, which defines another group of features, like scientific name, morphology and the capability to become extinct. Other instances of “Species” would be “Tiger”, “Crocodile” and “Dog”.

To refer to universals that have other universals as instances, UFO assigns the name of higher-order universals (hou). Notice that the universal recursion is unlimited. There are second-order universals, whose instances are first-order universals, but also third, fourth or even fifth-order universals, although these are rarely used. The closest concept to second-order universals in traditional conceptual modeling techniques is UML’s powertype, a class whose instances are other classes.

Another important definition in the foundational theory is that of a universal’s extension. It corresponds to a function that returns the set of all individuals that instantiate a universal in a given moment of time.

Furthermore, universals of the same order can be related to one another through generalizations (or specialization) relations. This particular type of binary relation defines a universal as the parent (or the super-type) of another universal, the child (or the subtype). By specializing a universal, one defines a subset of individuals that instantiate the parent universal and share some characteristics. From an extensional point of view, it always holds that every individual that instantiates a subtype also

stantiate the super-type. To exemplify, consider the first-order universals Person and Student. The latter is a subtype of the former because a student is a person enrolled in an educational institution.

Figure 1 illustrates these basics definitions in UFO: the generic category of things specialized in Universal and Individual. The rest of the concepts defined by the foundational ontology that we are going to present in this thesis specialize one of these two concepts.

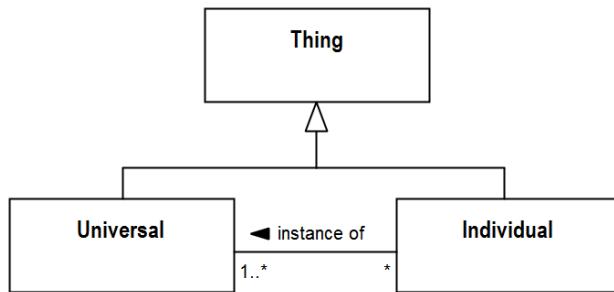


Figure 1. Basic UFO definitions.

Furthermore, Figure 2 depicts a layered perspective of universals and individuals. At the top, we represent the meta-conceptual level, which contains the universals defined in UFO, as the “Kind Universal” (we come back to its definition later). In the model level, we represent the specializations of Individual types. The figure illustrates two examples: the “Functional Complex” type, defined in UFO, and the “Person” type, defined in domain ontologies, i.e. in OntoUML models in general. Individuals, as Luke and Joe, which instantiate the universal “Person”, compose the last level, labeled as Instance Level.

The syntax of OntoUML, as defined in (GUIZZARDI, 2005) and updated in (ALBUQUERQUE; GUIZZARDI, 2013), only allow for the specification of first-order universals.

When creating a class or an association in OntoUML, a modeler is defining a new first-order universal. The embedded stereotype represents an instantiation relation between the formalized universal and the second order universal identified by the stereotype name. To improve readability, OntoUML suppress the name “universal” from the stereotype representation. Furthermore, the relation of “subtype of” between domains classes, like “Person”, and first-order universals of the foundational ontology

is implicit. We discuss the first-order universals detailed in UFO in the following sections.

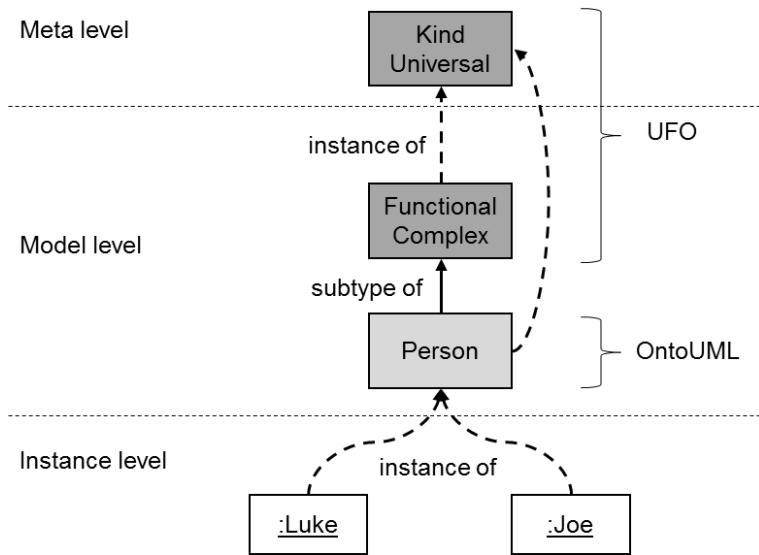


Figure 2. Universals, individuals and their relations in UFO.

2.2.2 Singular Individuals and Relations

The most basic classification of universals in UFO regards the distinction between *monadic* and *relation* universals. The former refers to meta-classes whose instances are first-order universals that can only be instantiated by singular individuals. For example, “Thesis” is a first-order universal that instantiates a monadic universal because its instances, like this work you are reading, are singular individuals. Conversely, relation universals are meta-concepts that apply to groups of two or more individuals, e.g. “being married to” is a first-order universal that instantiates this second-order universal.

Monadic universals are further specialized into *endurants* and *perdurants* universals, which aggregate classes that have endurant and perdurant individuals, respectively. Individuals are endurants if persist through time keeping their identity and are present as a whole whenever they are present (e.g. a ball, a tree). Perdurants, more commonly known as events, are the opposite: they are individuals that are composed of temporal parts, i.e., they do not exist in a moment, but happen throughout time (e.g. a fall, a football match)

Relation universals are refined into *formal* and *material* relation universals. The former is a meta-category applied to relations types that can hold between two individuals without the support of additional individuals. The latter category, conversely, only hold if there is an additional individual, the truth-maker of the relation. The instantiation and generalization relations are examples of formal relations. “Being married to”, is an example of material relation, since it only holds if an instance of marriage exists.

OntoUML’s design allows the representation of both monadic and relation universals: the former through the representation of classes and the latter through the representation of associations. Although the current OntoUML meta-model does not provide support for modeling events, researchers are investigating this extension (GUIZZARDI; WAGNER, 2012; MARTINS; DE ALMEIDA FALBO, 2008).

2.2.3 Basic Ontological Properties: Rigidity, Identity and Dependency

In addition to the basic aforementioned ontological categories, UFO defines some basic ontological properties whose combinations define the more concrete types of universals. UFO inherit these properties from OntoClean (GUARINO; WELTY, 2009).

UFO distinguishes universals according to their **rigidity** meta-property. Simply put, rigidity regards the necessity (in the modal sense) of individuals to instantiate given universals throughout time. *Rigid* universal are the ones whose instances necessarily instantiate them while existing, i.e., if an individual instantiate a rigid universal in a given moment, in every other moment it exists, it must also do so. Examples of rigid universals are Person, Car, Marriage, and Window. Conversely, *anti-rigid* universals are the ones whose instances contingently instantiate them, i.e., if an individual instantiate an anti-rigid universal, there is at least one possible moment in which the individual exists and do not instantiate it. Examples of anti-rigid universals are Student, Spouse, and Child. Lastly, UFO defines as *semi-rigid* universals the ones that behave as rigid for some individuals and as anti-rigid for others. To exemplify, consider a domain in which people are necessarily eligible for insurances (like life insurances), houses, however, are only eligible for insurances if their market value is superior to US\$ 100k. The universal Insurable Item, a common super-type of Person and

Insurable House, is semi-rigid because it is a necessary condition for people and an eventual one for houses, since their price may change.

The **identity principle** of an individual is a sort of “function” that asserts whether two individuals are the same or not. It is also, what allows one to count individuals. Sets (as in the mathematical notion) have one of the most simple identity principles: two sets are the same if, and only if, they contain the same individuals. Individuals must always follow a unique identity principle that cannot change through its existence. The nature of the relations between universals and identity principles of their instances differentiates universals: some aggregate individuals that follow the same identity principle, some aggregate others that follow different ones and some provide the identity principle for their instances.

Dependency is an ontological property that involves two first-order universals, u_1 and u_2 . One says that u_1 depends on u_2 , if every instance of u_1 must participate in a relation with instances of u_2 . In this work, the relevant dependency types are:

- specific dependency, characterized if while instantiating the dependent class, an individual cannot change the instance of the relation that characterizes the dependency (e.g. a living person has an specific dependency to her brain);
- existential dependency, a stronger version of specific dependency, but instead of instantiating, while existing, instances of the dependent class cannot change the instances of the universal they depend on (e.g. a person dependency to her brain is also existential – since the universal “Person” is rigid);
- generic dependency, a more relaxed form of dependency, in which every instance of the dependent class must participate in the relation that characterizes the dependency, but this relation instance might change throughout time (e.g. a person has an generic dependency to their hearts);

In the following subsections, we use these basic notions to define the more particular types of things that can exists according to UFO’s conceptualization.

2.2.4 Substantials and Moments

UFO refines endurants into *substantials* and *moments*, which applies on both the meta-class and the class hierarchies.

Moments (or *tropes*) are individuals that can only exist in other individuals (e.g. a headache, an intention, an object's weight), i.e. they are existentially dependent of the individuals they exist in, their bearers. Moments are the objectification of properties of individuals. Existential dependency is a necessary condition, but not sufficient for something to be a moment. Besides it, a moment must inhere in its bearer. Furthermore, moments inhere in exactly in one individual, even though, they can be existentially dependent on more than one.

Bearers can be moments themselves. For instance, the intensity of someone's headache is a moment, inhering in another moment, a headache, which inheres in a person. UFO prevents an infinite regression in the inheritance chain by requiring the chain to end on a substantial, another type of individual that does not inhere in any individual.

Substantials are endurants that do not inhere in other individuals. They have are highly independent individuals, like a table, a car or the portion of lemon juice inside a glass. Throughout this thesis, we also refer to substantials as objects.

2.2.5 Substantial Classification

Figure 3 presents UFO's classification of substantials. In the hierarchy on the left side of the figure, we present the meta-concepts, identifying with thicker borders those that have a direct mapping as OntoUML stereotypes. On the right side of the figure, we present the only three exclusive types of substances that exist according to UFO: functional complexes, quantities, and collections. Regardless of the types defined by domain ontologies, all substances individuals are instances of one of these following three types.

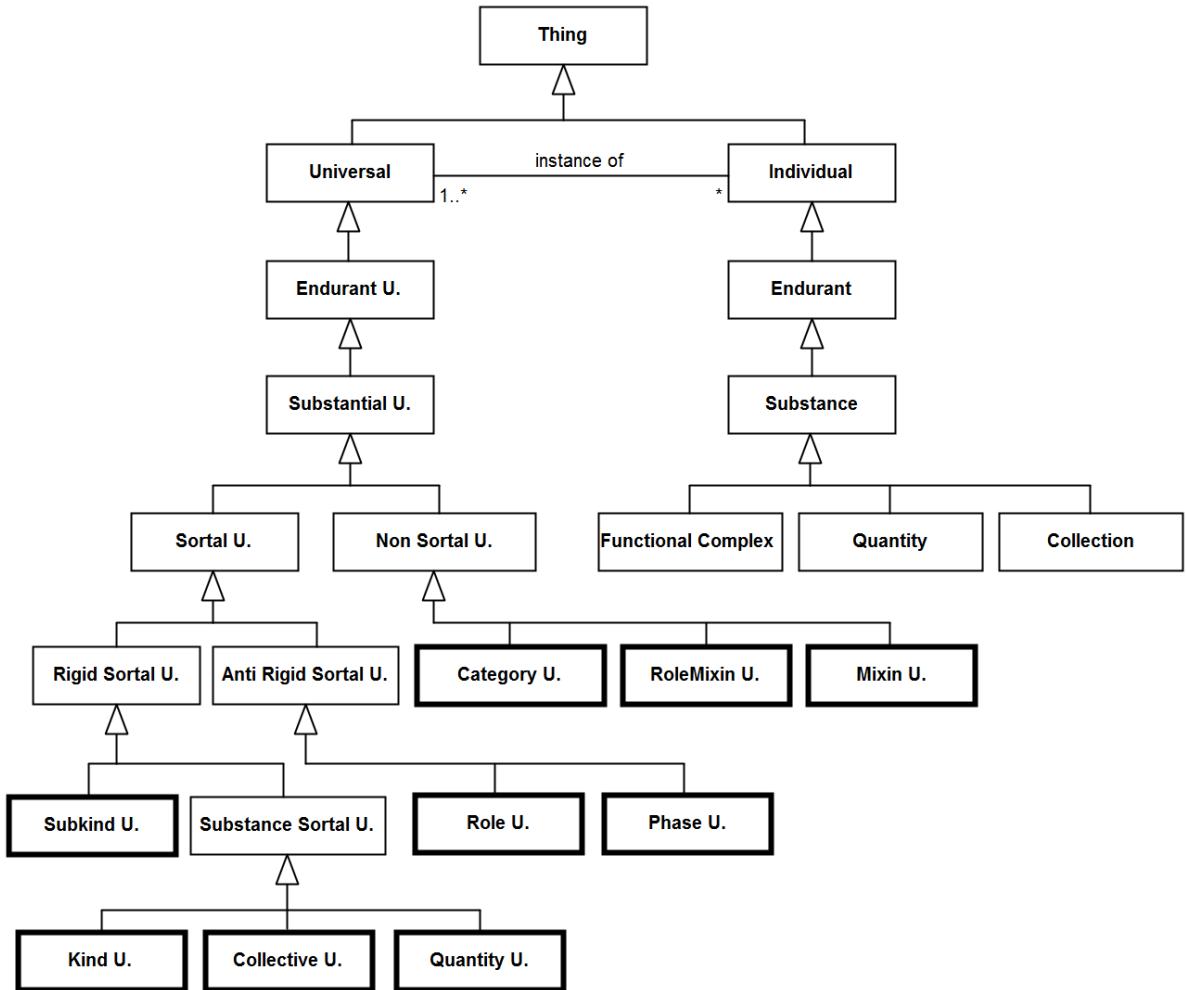


Figure 3. UFO's hierarchy of substantials.

The **Quantity** class stands for substantials that are maximal amounts of matter. It encompasses individuals with defined identity principles but undefined counting principles. Examples are the portion of coffee inside a cup, or the contents of a wine bottle. The **Function Complex** type stands for individuals that are composed by heterogeneous parts that contribute in different way to the function of the whole (e.g. a Car, which composed by motor, chassis, bumper, etc.). Lastly, **Collection** stands for individuals that have a homogeneous internal structure, in the sense that all its parts play the same role w.r.t. it, the one of being a part of it. From a linguistic perspective, mass nouns usually refer to quantities, whilst collective nouns usually refer to collections.

UFO's meta-universal hierarchy for substantials classifies the aforementioned types. The first level of refinement differentiates classes according to the identity principle of their instances. *Sortal Universals* aggregate classes whose instances follow the same

identity principle. Conversely, *Non-Sortal Universals* (or also known as Mixin Class Universals) qualify classes whose instances follow different identity principles. If we assume that chassis identify cars, and geographical and spatial dimensions identify real states properties, the Car and the Real State classes both fall into the sortal category. Now, if we consider the domain of an auction, which amongst other things, auctions houses and cars, the type Auctioned Item would be a non-sortal one.

Both *Sortal* and *Non-Sortal Universal* are specialized according to the rigidity meta-property. *Sortal Universal* is refined in *Rigid* and *Anti-Rigid Sortal Universals*. *Non-Sortals* are specialized into *Category*, which aggregates rigid non-sortal classes, *RoleMixin*, whose extension contains only anti-rigid non-sortal classes, and *Mixin*, which stands for semi-rigid non-sortal classes. These last three meta-classes have direct mappings to OntoUML stereotypes of analogous names.

Moreover, Rigid Sortal Universals can be further differentiate w.r.t. the characteristic of their classes to providing (or not) identity principles for their instances. Subkind universals stand for rigid sortal classes that do not define identity principles, whilst *Substance Sortal Universals* (or Ultimate Sortal Universals) are the ones that do provide identity for their individuals. Substance Sortals Universals are also distinguish by the nature of the instances of the classes they characterize. The *Kind Universal* aggregates first-order universals that refine the functional complex category. *Collective Universal* does that for collections and *Quantity*, for quantities.

Lastly, anti-rigid sortal classes can instantiate Role or Phase Universals. The former characterize classes whose instantiations are due to a change in a relational property, i.e., the establishment or destruction of a relation. The latter captures anti-rigid sortal types that qualify individuals accordingly to changes in intrinsic properties. For instance, the Student class is an instance of the role universal because people instantiate it when enrolling in an education institution. Child and Sick, on the other hand, are phases because they capture changes in intrinsic properties (i.e., moments).

2.2.6 Moment Classification

As for substantials, UFO also refines the definition of moments. Figure 4 depicts the hierarchy of moment classes, as well as its higher order counter-part. Once more, thicker lines highlight the meta-classes that give rise to OntoUML's stereotypes.

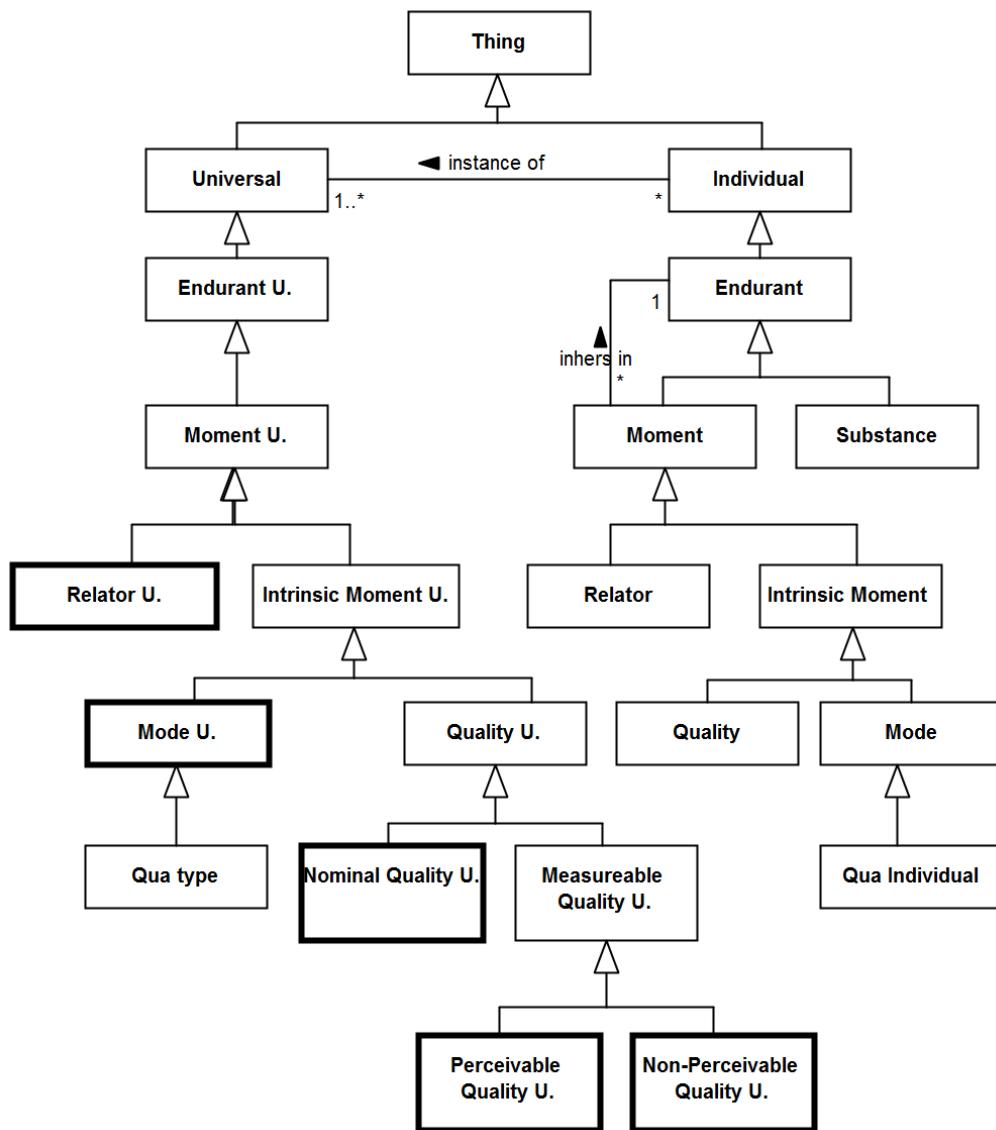


Figure 4. UFO's hierarchy of moments.

As we previously discussed, a moment is an individual that inheres in another individual, i.e., an objectification of an individual's property. *Relators* are moments that represent objectifications of relational properties, whilst *Intrinsic Moment* stands for moments that objectify intrinsic properties of the bearer.

The *Intrinsic Moment* class is further specialized into *Quality* and *Mode*. *Qualities* are objectification of properties that evaluate into a certain value space. Mass is an example of quality, which can be evaluated in terms of a positive numeric value in a kilogram or pound scale. Other examples are height, volume, color, name, date. *Modes*, conversely, represent unstructured intrinsic properties, like someone's intention to do something or belief in something.

UFO distinguishes qualities according to their measurability (ALBUQUERQUE; GUIZZARDI, 2013). A *Nominal Quality* refers to social conventions and provides a name for an individual, like birth names, passport numbers, postal codes, etc. One cannot measure a nominal quality in a scale because they are just lexical combinations that follow a determined structure. *Measurable Qualities*, conversely, are the ones whose values fall into a scale. Examples are the currency value, a scale of positive decimal values, and the weight of a car, measured in grams, pounds, tons and so on.

Furthermore, UFO classifies qualities with respect to the capability of cognitive agents to measure them. Some qualities, usually physical ones, like color, length, and weight, are directly perceivable by sensorial apparatus, thus referred to as *Perceivable Qualities*. Others, like the currency value of an asset, are not. UFO classifies the latter type as *Non-Perceivable Qualities*.

The *Qua Individual* is a particular type of Mode. They refer to unstructured moments that an individual acquires when in a particular context, usually when playing roles. For instance, consider the individual Barack Obama. While playing the role of the president of the United States of America, he acquires a qua-individual, Obama-Qua-President, which provides him a set of powers and duties, like attending determined events. Moreover, we can think of him as a father, Obama-Qua-Father, which entails him different properties, like having a child and legally answering for his offspring.

Coming back to the Relators, the objectification of relational properties (or their truth-makers), are in fact composed by relational qua-individuals. Take for instance the relation of being married to, which holds between two people. If we say that Barack Obama is married to Michelle Obama, it is because there exists an instance of the relator Marriage composed by Michelle-Qua-Wife-Of-Obama and Obama-Qua-Husband-Of-Michelle.

2.2.7 Relation Universals

Figure 5 depicts UFO's hierarchy of relation universals. Differently from substantials and moments, we only represent the universals as classes, since the corresponding individuals would be represented as lines, associations in our UML-like syntax. Once more, we represent the relation universals that can be directly instantiated in OntoUML models with thicker borders than the rest.

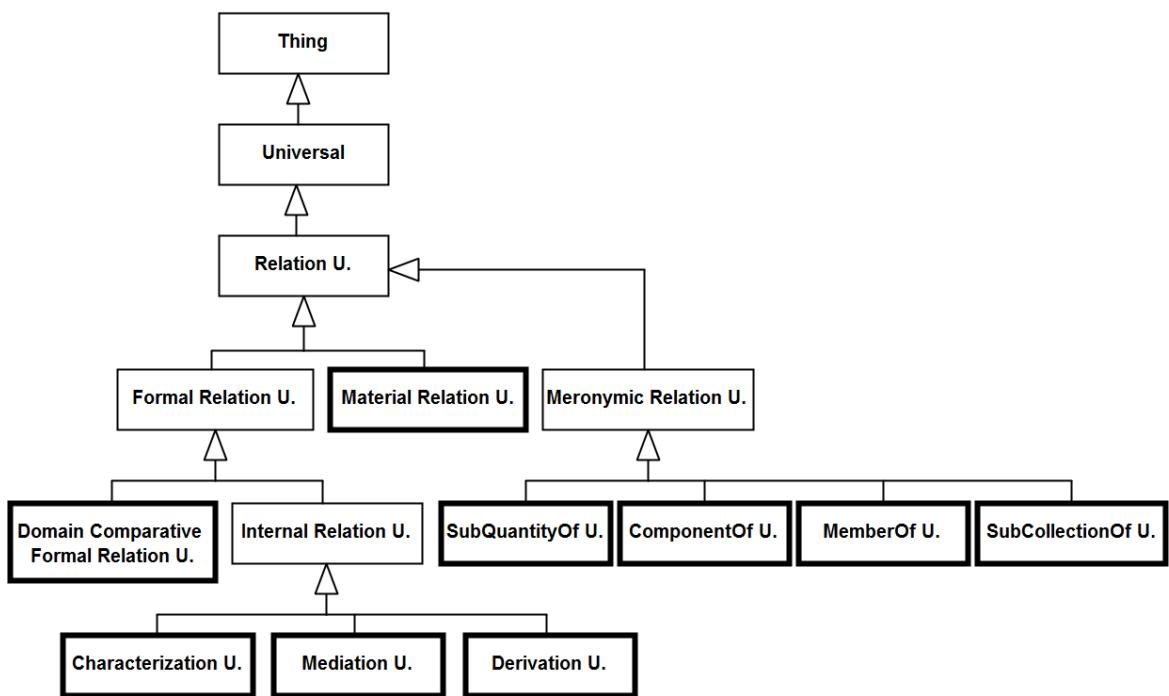


Figure 5. UFO's hierarchy of relation universals.

The first distinction between the types of relations regards the conditions required for them to hold. Material Relation Universals capture types of relations that require additional entities to hold, i.e. instances of relator types. The relation “researching at”, which holds between a researcher and a research institution, is classified as material because it requires a social agreement for it to be true, like an employment contract. Opposed to the material relations, there are the formal relations, which can hold between individuals despite the existence of other entities.

UFO further classifies formal relations as internal and external, whether they imply or not in existential dependency between the related types (in one of both directions). The only external relation described in UFO is the *Domain Comparative Formal Relation* (*DCFR*), which stands for relations that are reducible to the comparison of values of

qualities that characterized the related individuals. For instance, the relation “being taller than” is a DCFR because it can be reduced to the comparison of the height property of two individuals. Internal relations, conversely, are not reducible in this way. There are three types of internal relations in UFO. The first, labeled as *Characterization*, stands for the inherence relation that holds between moments and the individuals they characterize. The second, named *Mediation*, represents the relations between individuals and the truth-makers of material relations. Lastly, *Derivations* represent the relation between Material relations and their truth-makers, called Relators.

Orthogonally to the formal and material classification, relations can be *Meronymic*, or as more commonly known in by the conceptual modeling community, part-whole relations. Despite further classifications, every part-whole relations obey a set of additional axioms:

- *weak supplementation*, which states that every whole must be composed by at least two parts;
- *irreflexivity*, individuals cannot be a part of themselves;
- *asymmetry*, if ‘a’ is part of ‘b’, ‘b’ cannot be part of ‘a’;
- *acyclicity*, an individual cannot be in its part-hood transitive closure (part of its parts, or parts of parts of its parts, and so on).

Moreover, all meronymic relations have the following additional Boolean meta-properties:

- *isEssential*, which implies an existential dependency from the whole to the part;
- *isInseparable*, which captures an existential dependency from the part to the whole;
- *isImmutablePart*, a specific dependency from the whole to the part;
- *isImmutableWhole*, a specific dependency from the part to the whole;
- *isShareable*, a boolean meta-property that, when set to true, forbids an individual to compose more than one whole of the same type.

UFO defines the particular types of meronymic relations proposed in UFO based on the type of individuals that can act as the whole, and as the part, combined with some particular values for the meta-properties.

The *ComponentOf Universal* stands for meronymic relations that hold between functional complexes (e.g. the relation between a car and its engine, the relation between the human body and a heart). By default, no meta-property value is set. Furthermore, the *MemberOf* relation Universal stands for part-whole relations defined between Collections and their members (e.g. Band-Member, Puzzle-Piece, Wolf Pack-Wolf), i.e., a membership relation. *MemberOf* relations can hold between collections (as the whole) and functional complexes (as the members) or between two collections. The *SubCollectionOf* universal qualifies relations that hold between collections and their sub-collections (e.g. Deck-Heart Cards, Amazon Forest-Brazilian Part of the Amazon Forest). Lastly, there is the *SubQuantityOf* relation universal, which stands for part-whole relations that hold between quantities (e.g. Beer-Water; Concrete-Sand). By default, the *SubQuantityOf* relation is inseparable and non-shareable.

2.3 QUALITY CRITERIA FOR ONTOLOGIES

To assess the quality of an ontology, we must understand it as an engineering artifact. By viewing ontologies as artifacts, we consider them as something intentionally made by agents, envisioning a subsequent use or purpose. Therefore, a good ontology is one that is adequate for its intended uses.

To guide modelers in systematically producing good ontologies, the research community proposed through time, many different measurements or criteria. Vrandečić presents a survey in (2009), in which he compares and consolidates a common set of quality criteria. These criteria regard different aspects of an ontology, such as its syntax, semantics, usability, understandability and so on. Although we understand the importance of all these different aspects of ontology quality, we focus in this thesis on the semantic dimension. That means that we are mostly concerned on how faithful a formalization is to the way a community understands a portion of reality.

We use four semantic measurements to guide the development of the validation framework we propose in this thesis:

- **precision:** measures if the ontology has problems of under-constraining, i.e., if it allows instantiations that were not intended by the modeler;

- **coverage:** measures if the ontology has problems of over-constraining, i.e., if it does not allow desired instantiations;
- **scope:** measures if the ontology formalizes every concept, property and relationship required to explain a domain, and only them;
- **classification:** measures if the modelers chose the appropriate categories of the foundational ontology do describe the domain entities;

Precision and coverage are measurements built upon the notions of intended and possible instantiations of an ontology (GANGEMI et al., 2005). On one hand, intended models refer to all possible ways a modeler wants her ontology to be instantiated. More technically, the set of admissible states of affairs defined by the conceptualization the ontology is capturing. On the other hand, possible instantiations refer to all valid instantiations of the ontology. Therefore, intended instantiations are “things” the modeler wanted to say and the possible instantiations the “things” that she actually said.

Precision refers to the amount of unintended instantiations, i.e. if the model “allows” more instantiations than desired. It is the ratio between intended instantiations by all possible instantiations. To exemplify, consider the “married to” relation, which holds between two people. In a common sense ontology about marriage, if a person can marry herself, the ontology is not precise.

Coverage regards whether the model allows all intended instantiations. To exemplify, consider again the “married to” relation. If my community understands that a marriage can hold between people of the same gender, as well as between people of different ones and my ontology only allows the latter to happen, it has coverage issues.

Precision and coverage, therefore, identify problems of under-constraining or over-constraining in an ontology, respectively. Adapted from (GANGEMI et al., 2005), Figure 6 classifies how good an ontology is only by evaluating these two quality criteria.

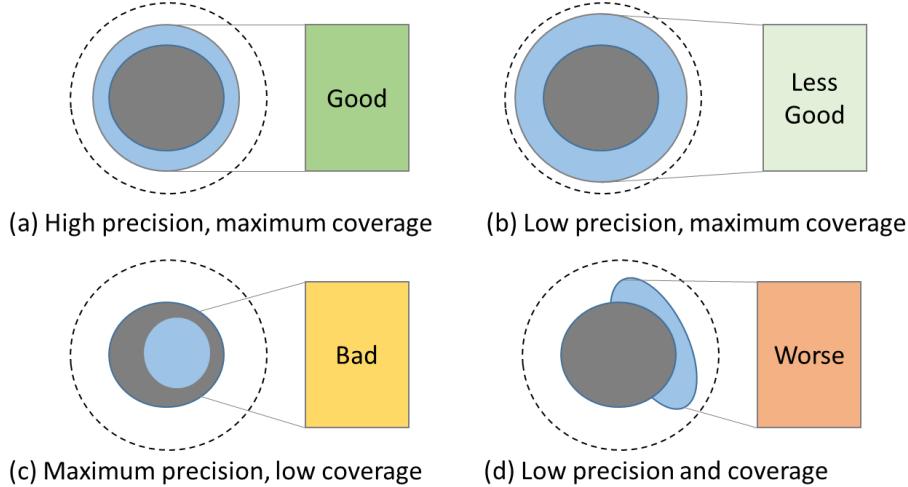


Figure 6. Intended and possible model instantiations, adapted from (GANGEMI et al., 2005).

Scope measures if the modeler managed to capture all the concepts, properties and relationships that are relevant for describing a given domain of interest, but only those. As an example, consider a common sense ontology about cars. The ontology would have scope problems if it describes a car having only wheels as parts, leaving out the bumper, the windshield, the engine and so on.

The classification criteria assess how adequate are the modelers choice to represent a concept in terms of the meta-categories of the foundational ontology being used. For instance, consider the concept of car and the “married to” relation. According to UFO, a car is a functional complex, because it is something that persists through time keeping its identity and its parts play different functions regarding it. In addition, the “married to” relation should be classified as material according to UFO, because it only holds between people if, in the past, an event occurred that lead to the creation of a relator between them: the marriage. If a modeler represents a car as a quantity or the “married to” relation of formal, it indicates an ontological classification issue.

We are not claiming here that there is only one way to classify concepts according to a foundational ontology. We agree that different communities might classify the same terms in a different way. The goal of the ontological classification criteria is to identify inconsistent decisions.

2.4 ONTOLOGY VALIDATION

In order to improve the overall semantic quality of an ontology, an appropriate language, based on a foundational ontology, should be used to design the model (GUARINO, 1998). Nonetheless, it is not enough to guarantee that modelers will only produce good models. To assert that, one would have to believe that users make no mistakes while constructing the model and trust that they fully understand the theories that support the modeling language.

In order to improve the semantic quality of models in a systematic way, users should perform validation. We understand validation as an activity that aims to improve the semantic aspect of the model, making it closer to the intended conceptualization. Many approaches have already been proposed for this end, such as model simulation (BENEVIDES et al., 2010a), model testing (TORT; OLIVÉ, 2010) and the application of formal methods (QUERALT; TENIENTE, 2012).

A modeler can say that she validated her ontology if she has the confidence to say: "*I built the right model for this domain*". In order for a modeler to mean that truly, she must assess a number of quality criteria. A validated ontology must cover all relevant concepts within the shared conceptualization (scope). Furthermore, the user must classify the formalized concepts using the appropriate ontological meta-categories (classification). Moreover, the ontology must not allow instantiations that do not correspond to admissible states of affairs (precision), at the same time that all admissible states should refer to ontology instantiations (coverage).

At this point, we want to point the difference between verification and validation, since the community uses these terms with many different names, and sometimes even as synonyms. We understand these as complementary activities that should be performed in a model, but which aim to identify completely different types of problems. As we said, validation is concerned with the semantic dimension of the ontology. Verification, conversely, refers to the syntactical part. Therefore, we understand verification as the activity of checking if one specified the ontology following all the syntactical constraints of the adopted modeling language. Although we see the verification as an important first step, syntactical errors are out of the scope of the research presented in this thesis.

3 REVISITING ONTOUML2ALLOY

Conceptual model simulation is an approach to automatically generate model instances and check properties through the usage of lightweight formal methods, e.g. Alloy (JACKSON, 2012). The assumption is that by visually inspecting possible instantiations, a modeler can sort out admissible and non-admissible model instantiations. By recurrently performing this analysis, modelers improve model precision. In addition, modelers can also “demand” the generation of particular model instances that are expected to hold. If she encounters no model instance, her model has an over-constraining problem.

Model validation through the usage of Alloy has been initially proposed for UML (BORDBAR; ANASTASAKIS, 2005). Inspired by them, the approach has been applied to OntoUML (BENEVIDES et al., 2010b; BRAGA et al., 2010). Although these authors propose to use Alloy, a lightweight formal method, their proposal is not as “user friendly” as it might seem. One should not understand lightweight formal method as an “easy to use” method.

This chapter revisits the proposed transformations from OntoUML to Alloy to make the validation approach more accessible for “managers”, i.e., conceptual modelers that are not expert in logics and formal methods. We accomplish that by providing guidance on how to use the simulation capabilities, that is, relating desired model properties to pre-defined test cases, proposed as validation scenarios.

We start by motivating the need to redefine the mappings of the original transformations from OntoUML to Alloy. In the following, we specify the new mappings and compare to the previous. Lastly, we define and discuss the simulation scenarios, detailing the types of properties tested by them. For a detailed explanation of the Alloy language, refer to Annex A.

3.1 WHY REVISIT ONTOUML2ALLOY?

We revisit the transformation from OntoUML to Alloy because of multiple reasons. It starts with the evaluation reported in (SALES, 2012), in which a series of tests are performed to evaluate the transformation rules and the tool support of Benevides' and Braga's proposals. That report identifies coverage issues in both proposals, especially regarding meta-attributes, like *readOnly* (used to formalize specific dependencies) or the enforcement of the weak supplementation axiom on part-whole relations (a whole to be composed at least by two distinct parts). In addition, the report identifies that some transformation rules did not work as anticipated. That is the case for the transformation of OntoUML categories into Alloy functions, which syntactically precludes the specification of relations to categories, and the implicit transformation of generalizations, which makes multiple inheritance unfeasible in some cases. Lastly, the report lists many implementation bugs, which results in both syntactically and semantically problems in the generated Alloy specification.

Furthermore, we argue that both proposals adopt a demanding assumption: users only simulate syntactically correct OntoUML models. In fact, we claim the exact opposite: users will most likely provide partial and/or wrong models. Thus, the design of a sound model transformation must anticipate and cope with difficulties that may arise from ill-defined models.

We refer to *partial* OntoUML models as the ones that have syntactical issues but one can still be fixed by adding new elements to them. An example is a role class specified without its characterizing relational dependency (mediation). Another example is a class stereotyped as subkind, which does not inherit any identity from an ultimate sortal (kind, quantity or collective). We claim that users are likely to validate partial models. Firstly, because OntoUML models grow quickly in size and one must separate relevant fractions to be able to understand the results of the simulations. We do not assume that every relevant model fragment is syntactically valid. Secondly, because the simulation technology itself poses a size limitation, thus users are required to crop their models in order for the analyzer to be able to provide the results. Finally, because users can, on purpose, partially define models. For example, the core ontology of services, named UFO-S (NARDI et al., 2013), specify only mixins and no sortals. This

lack of identity providers generates many syntactical errors, but they are all intentional, since the authors designed the ontology to be a generic reference model, which should not comprise the identity of individuals.

By wrong models, we mean those that cannot become syntactically valid only by adding new elements. At least one element must be deleted or have its stereotype changed (in the case of classes and associations). An example of a wrong model is one that has a class stereotyped as kind specializing a class stereotyped as relator. The only ways to rectify such model is to change the stereotype of the classes or remove the generalization between them.

The argument to anticipate simulations of models with such nature is two-fold. First, we argue that no modeling language will cover all modeling necessities for formalizing all concepts in every domain, even sophisticated languages like OntoUML, which complies with a foundational ontology. An example of such shortcomings, in OntoUML, is the lack of support for representing higher-order classes (e.g. Species, Car Model), i.e. classes whose instances are other classes. Whenever modelers are confronted with problems like this, they either accept the limitation and do not model the concept, or (what is more likely) propose a work-around, i.e., use the language in an unexpected way to cope with the modeling problem. The problem is that, solutions of this nature are not always syntactically valid. So, if a simulation is exclusively accepts valid models, we limit its usage in cases like this. The evidence encountered in the models we gathered during this research (see Section 6.1), substantiates the claim that syntactical errors are recurrent.

The second argument to support the simulation of models that cannot become syntactically valid only by adding new elements is that the Alloy specifications generated from OntoUML models should be as faithful as possible from its source. A logical inconsistency in the original model should generate the exact same inconsistency on the produced model. This way, modelers can directly assess the logical consequences of their decisions. For example, OntoUML assumes that every object must have an identity and that kinds are identity providers. Thus, all kinds in a model are automatically disjoint, i.e., no individual can ever instantiate more than one kind. Now, if one specifies a kind as a subtype of another kind, it generates a logical inconsistency on the child class, which will have an empty extension. We argue that

this same problem inconsistency should appear in the Alloy specification, instead of forbidding the user to simulate.

The last reason we revisit the OntoUML to Alloy transformation is because they are designed and implemented using an outdated OntoUML meta-model, proposed by Benevides (2010). The new version, called RefOntoUML, is proposed by Carraretto (2010), who points some limitations regarding Benevides' original metamodel implementation, like its incompatibility with the rules for constructing UML 2.0 profiles, its lack of support for some language constructs and the undesirable mixture between concrete, and abstract syntax. Carraretto's meta-model has also been updated to provide support for modeling qualities (ALBUQUERQUE; GUIZZARDI, 2013), and is currently the basis for all development of conceptual modeling tools in NEMO.

3.2 ONTOUML2ALLOY 2.0

In this section, we present the refactored structure of the transformation from OntoUML to Alloy. In Section 3.2.1, we present the adopted World Structured inherited from Benevides' proposal (2010b). In Section 3.2.2 we discuss the Alloy module that contains the predicates to enforce most ontological properties defined in the OntoUML, like rigidity and dependency. Moreover, Section 3.2.3 describes the skeleton structure of the Alloy specifications we generate. In the sequence, Section 3.2.4 details every step of the transformation, i.e. what is the corresponding specification pattern in Alloy of each OntoUML construct. Lastly, in Section 3.2.5, we discuss the parameters for the new transformation, introduced to cope with the challenges of simulating partial models.

3.2.1 World Structure Module

The world structure we use in our transformation is the one proposed in (BENEVIDES et al., 2010a). For the sake of simplicity, we refrain from making repeated references to this work in this Section.

A world represents a possible instantiation of the ontology in a given moment, i.e. a model snapshot. It is the representation of a possible state of affairs according to the conceptualization formalized by the ontology. An existence relation relates a world to individuals, i.e. individuals exist in worlds. Within a world, an individual can instantiate types and be related to other individuals through the instantiation of associations. We refer to the set containing all individuals that exist in a world as its *population*.

Succession relations interconnect Worlds. The real world meaning of a world being a successor of another is that, from a given state of affairs, identified by the first world, a sequence of events can occur, leading to the second world. Notice that in the proposed branch structure, every world can have at most one predecessor, but any number of successors. Furthermore, a world cannot be directly or indirectly successor of itself, making the branching structure resemble an n-ary tree.

A world branch is a set of worlds that succeed one another. The structure resembles a directed graph, in which the worlds are nodes, and the accessibility relations are the edges. World branches have an additional constraint that requires every word in the branch to be accessible or access any other, directly or indirectly.

The structure comprises four world categories: Past, Future, Counterfactual and Current worlds. A current world stands for the current state of things, an analogy to the present time. Future worlds present possible state of affairs that can become true if we continue to move through time from the current world. Past worlds, conversely, are the ones were true and led to the current world. They present the outcomes of a series of events that lead to the current situation. Finally, the counterfactual worlds depict situations that could have happen if something went differently in the past.

Every generated branch in the simulation must have exactly one current world, randomly chosen by the simulation. The others worlds are classified relatively to it. If the current world is a direct or indirect successor of a given world, it is classified a past world. Conversely, Future Worlds are direct or indirect successor of the Current World, signature. Lastly, a Counterfactual World is one that does not lead to the current world and follows from it. Instead, it is an alternative future of a past world.

The last constraint imposed on the world branch structure is the continuous existence. It forbids an individual from existing again in a branch if it ceases to exist at one point.

To exemplify, consider a linear branch composed by w_1 , w_2 and w_3 , such that w_1 leads to w_2 and w_2 leads to w_3 . If an individual x exists in w_1 , but ceases to exist in w_2 , it cannot exist again in w_3 . The *continuous_existence* predicate enforces this constraint in the world structure module.

We define a constraint not on the branches themselves, but on the relation between individuals and the worlds in which they exist. Moreover, the predicate *elements_existence*, states that every individual must exist in at least one world.

The representation of the aforementioned world structure in Alloy follows in Listing 1.

Listing 1. World Structure module in Alloy.

```

1  module world_structure[World]
2
3  some abstract sig TemporalWorld extends World{
4      next: set TemporalWorld -- Immediate next moments.
5  }{
6      this not in this.^(@next) -- There are no temporal cicles.
7      lone ((@next).this) -- A world can be the immediate next moment
8      of at maximum one world.
9  }
10 one sig CurrentWorld extends TemporalWorld {} {
11     next in FutureWorld
12 }
13
14 sig PastWorld extends TemporalWorld {} {
15     next in (PastWorld + CounterfactualWorld + CurrentWorld)
16     CurrentWorld in this.^@next -- All past worlds can reach the
17     current moment.
18 }
19
20 sig FutureWorld extends TemporalWorld {} {
21     next in FutureWorld
22     this in CurrentWorld.^@next -- All future worlds can be reached by
23     the current.
24 }
25
26 sig CounterfactualWorld extends TemporalWorld {} {
27     next in CounterfactualWorld
28     this in PastWorld.^@next -- All past worlds can reach a
29     counterfactual
30 }
31
32 --Objects cease to exist and come back in the future
33 pred continuous_existence [exists: World->univ] {
34     all w: World, x: (@next.w).exists | (x not in w.exists) => (x
35     not in ((w.^next).exists))
36 }
37
38 --All elements must exists in at least one world
39 pred elements_existence [elements: univ, exists: World->univ]{
40     all x: elements | some w: World | x in w.exists
41 }
```

3.2.2 Ontological Properties Module

In order to characterize an OntoUML model as an Alloy specification properly, we defined another auxiliary module, which builds upon the World module presented in the last section. We define seven predicates, namely: *antirigidity*, *rigidity*, *elements_existence*, *immutable_target*, *immutable_source* and *derivation*.

Listing 2 provides the complete specification of the ontological properties module.

Listing 2. Ontological Properties module in Alloy.

```

1  module ontological_properties[World]
2
3  /*run this predicate to verify if there the class is actually anti-
4  rigid. Parameterized the predicate with the class name.*/
5  pred antirigidity[class:set univ->univ, class_type: univ, exists:univ-
6  >univ] {
7      all x:class_type | #World>=2 implies (some disj w1,w2:World | x
8      in w1.exists and x in w1.class and x in w2.exists and x not in
9      w2.class)
10 }
11
12 /*this predicate states that a class is rigid.*/
13 pred rigidity[class: univ->univ, class_type: univ, exists:univ->univ]
14 {
15     all w1:World, p:univ | p in w1.exists and p in w1.class implies
16     all w2:World | w1!=w2 and p in w2.exists implies p in w2.class
17 }
18
19 /*predicate to make the target association end is immutable*/
20 pred immutable_target [source_class:World->univ, assoc: univ->univ-
21 >univ] {
22     all w1:World, x:univ | x in w1.source_class implies all w2:World
23     | x in w2.source_class implies x.(w1.assoc)=x.(w2.assoc)
24 }
25
26 /*predicate to make the target association end is immutable*/
27 pred immutable_source [target_class:World->univ, assoc: univ->univ-
28 >univ] {
29     all w1:World, x:univ | x in w1.target_class implies all w2:World
30     | x in w2.target_class implies (w1.assoc).x = (w2.assoc).x
31 }
32
33 /*states that the material relation is derived from the relator*/
34 pred derivation [material: World->univ->univ->univ, mediation1: World-
35 >univ->univ, mediation2: World->univ->univ], relator : World->univ,
36 mediated_class1:World->univ, mediated_class2:World->univ] {
37     all w: World, x: w.mediated_class1, y: w.mediated_class2, r:
38     w.relator | x -> r -> y in w.material iff x in r.(w.mediation1) and y
39     in r.(w.mediation2)
```

The *antirigidity* predicate does what its name suggests: it enforces the exemplification of anti-rigidity for a given class. In natural language, the predicate means that every individual that instantiates an anti-rigid class in any world must exist at least in one other world and do not instantiate the class (if the world branch has more than one world). Naturally, this predicate only affects the simulations if the scope of the World signature is greater than one.

In the same way, the *rigidity* predicate characterizes rigid types. The predicate states: for all worlds and individuals, if an individual x exists in a world w and it is in the extension of a class c , then for every other worlds w' , such that w is different than w' , x existing in w' implies that x is in the extension of c .

The predicates entitled *immutable_source* and *immutable_target* formalize specific dependencies in each direction of an association. The *readOnly* meta-attribute (owned by the Property meta-class on the meta-model) and the *isEssential*, *isInseparable*, *isImmutablePart* and *isImmutableWhole* meta-attributes (owned by part-whole relations) capture specific dependencies in OntoUML models. Even though there are distinctions of specific and existential dependencies (the latter is a “stronger” version of the former), we do not specify other predicates. This design decision comes from the fact that specific dependency combined with rigid dependent classes implies in existential dependency. Furthermore, anti-rigid existentially dependent types imply in “pseudo” rigid dependent classes. Therefore, adding more predicates would only make the transformation more complicated and add no additional expressivity.

The predicate that formalizes specific dependency means that for every instance x of the dependent class that exists in a world w , for all world w' , if x exists and instantiates the dependent class, the individuals to which x is connected to in w are equal to the ones x is connected to in w' .

We also define the *derivation* predicate. It formalizes the derivation of material relations from relators. Its definition reads, in natural language, that a material instance (defined by a triple of source class \times relator \times target) only occurs if, and only if, an instance of a mediation connecting the source class to relator exists, alongside another connecting the target class to the relator.

3.2.3 Skeleton Specification

Listing 3 illustrates the common structures we include in every Alloy specification generated according to our transformation from OntoUML to Alloy.

Listing 3. Skeleton structure of a generated Alloy module.

```

1  module Model
2
3  open world_structure[World]
4  open ontological_properties[World]
5  open util/relation
6  open util/sequniv
7  open util/ternary
8  open util/boolean
9
10 sig Object {}
11 sig Property {}
12 sig DataType {}
13
14 abstract sig World {
15     exists: some Object+Property,
16 }
17
18 fact additionalFacts {
19     continuous_existence[exists]
20     elements_existence[Object+Property,exists]
21 }
22
23 fun visible : World->univ { exists }
24
25 run { } for 10 but 3 World, 7 int

```

The first line, `module Model`, is the module declaration. Every instance of the OntoUML meta-model starts with one element name that act as the root of all other elements. It can be either a RefOntoUML::Model or a RefOntoUML::Package. In both cases, its name defines the name of the produced Alloy module.

Following, we declare six module importations in sequence. The first, `open world_structure[World]`, is the importation of the module that contains the world structure and the relation between worlds, as described in section 3.2.1. The second, `open ontological_properties[World]`, is the importation of the ontological properties module, which is required to use the predicates listed in section 3.2.2. The last four are importation of Alloy's standard utility libraries. The line `open util/relation` imports the library that contains predicates to specify binary properties, like reflexivity and transitivity. The next line, `open util/sequniv` imports a library that enables the use of ordered relations (required for the implementation of the *isOrdered* meta-attribute).

Further, the line `open util/ternary` imports module containing functions to manipulate triples, which we will use for handling material relations. At last, we write `open util/ternary`, which imports the Boolean module, require in the transformation of Boolean attributes.

After the module importations, there are three signatures declarations. The first, `sig Object{}`, is only created if the inputted model has at least one object class, i.e., one class stereotyped as Kind, Collective, Quantity, Role, Phase, RoleMixin, Mixin or Category. It represent the set of all object individuals, regardless of the types it instantiates. Analogously, the declaration `sig Property{}` is only created if there is at least one moment class in the model, i.e., stereotyped as Relator or Mode. It also comprises the set of all properties, regardless of the types they instantiate. Finally, the line `sig DataType{}`, is created if any custom datatype is specified (integers, for example, are mapped into Alloy's default corresponding type).

The only signature that will appear in every generated Alloy specification is the World signature. The declaration `abstract sig World` contains states that the worlds is abstract so every world will be either a Past, Future, Current or Counterfactual world, as defined in the world structure module. This signature always contains the field declaration `exists: some Object+Property`, which defines the relation between individuals (objects and properties) and the worlds in which they exist. Note that the word "some" precedes the field declaration. This means that every world must have something existing in it. In addition, we intentionally excluded the Datatype signature from the field declaration because we adopt the vision that data types are atemporal entities.

Additionally, we declare a fact block named `additionalFacts`. It serves the exclusive purpose of calling two predicates already discussed in the world structure module: `continuous_existence[exists]`, prohibiting elements to cease to exist and then come back to life; and `elements_existence[Object+Property,exists]`, requiring every individual to exist in at least one world.

In the sequence, `fun visible : World->univ { exists }` declares a function that returns all things that exist. This declaration has no effect on the simulation what so

ever, it is just a workaround we kept from (BRAGA et al., 2010) to facilitate the customization of the Analyzer’s visualization tool used to show simulation results.

The last line, `run {} for 10 but 3 World, 7 int` is just a command to generate a unrestricted simulation with at most 10 atoms of the signature Object, 10 atoms of Property and 10 atoms of DataType distributed along 3 atoms of signature World. The expression “7 int” refers to the range of integers considered in the simulation: from 2^{-7} to 2^7-1 .

3.2.4 Transformation Rules

In the following subsections, we detail our mappings of OntoUML elements to Alloy expressions. We present the mappings grouped by the type of OntoUML element they are related to.

3.2.4.1 Classes

We map every class, regardless of its stereotype, into a field declaration within the World signature, as shown in Table 1. Each class-generated field relates every world to a subset of things that exist in that world. If the transformed class is an object (stereotyped as kind, quantity, collective, subkind, role, phase, roleMixin, mixin or category) the field’s type is the projection of the Object signature, as in `ObjectClass: set exists:>Object`. Conversely, we translate classes stereotyped as relator or mode into fields that map to the projection of the Property signature, as in `PropertyClass: set exists:>Property`. Notice that by declaring each field with the `set` multiplicity, we state that it is optional for world to have an instance of a class.

To distinguish between rigid, anti-rigid and semi-rigid classes, we use the rigidity predicates discussed in the ontological properties module. For every rigid class, stereotyped as kind, collective, quantity, subkind, category, relator, mode or quality, we call the rigidity predicate, as in `rigidity[RigidClass, Object, exists]`. The first parameter corresponds to the class name, the second, to the signature used in its definition (Object or Property) and the third is always world’s “exists” field. For every

anti-rigid class, we only call the anti-rigidity predicate if the corresponding parameter is set to true (see more about transformation parameters in section 3.2.5). We enforce anti-rigidity by calling the predicate `antirigidity[AntirigidClass, Object, exists]`. Lastly, semi-rigid classes do not require any further constraining.

Table 1. Class mapping to Alloy.

OntoUML	Alloy
ObjectClass, PropertyClass	<pre>abstract sig World { ObjectClass: set exists:>Object PropertyClass: set exists:>Property }</pre>
RigidClass	<pre>fact rigid { rigidity[RigidClass, Object, exists] }</pre>
AntirigidClass	<pre>fact antirigid { antirigidity[AntirigidClass, Object, exists] }</pre>
<i>isAbstract</i>	<pre>fact abstract { all w: World w.AbstractClass = w.Subtype1 + w.Subtype2 + ... + w.Subtype-n }</pre>
-	<pre>-- additional facts abstract sig World {} { exists:>Object in ObjectClass1 + ... + ObjectClass-n exists:>Property in PropertyClass1 + ... + PropertyClass-n }</pre>

In summary, we represent classes as binary relations between worlds and the corresponding set of individuals, Object for substantials and Property for moments. Thus, the expression `World.Class` refers to every individual that instantiate the class in any world, whilst the expression `w.Class` (`w` being a particular World), returns the individuals that instantiate Class in `w`.

3.2.4.2 Associations

In general, we map an association between the classes “Source” and “Target” as a ternary relation between World, the source and target classes, as in `association: set Source -> Target`. We declare all association fields within the World signature. As we

did for classes, we always declare associations using the “set” constraint, in order make it optional for a world to have an instance of the association.

If a material association, however, is derived from a relator (through a derivation), the mapping is slightly modified. In these cases, we do not map the material relation as a triple, but as a quadruple, the fourth dimension being the relator the truth-maker of the material relation. Thus, a material association between classes “Source” and “Target”, whose truth-maker is “Relator”, generates the following field declaration: `material: set Source -> Relator -> Target.`

Amongst all types of associations, derivations are the only ones that we do not map into fields within the World signature. In order to translate a derivations embedded meaning, we generate a fact to correlate the instantiation of the corresponding material to the instantiation of the mediations connected to the respective relator. Table 2 shows the mapping.

Whenever an association contains at least one end set as ordered (`isOrdered=true`), the general mapping into ternary fields must be transformed into a 4-ary field, just like for derived material associations. This time, instead of adding a relator as the fourth dimension, we add Alloy’s primitive “Int” signature, which generates the following declaration: `association: set Source set -> set Int set -> set Target.` The integer dimension provides the “position” of the relation. Adding the integer dimension, however, is not enough to characterize an ordered relation. To do that, we add the following expression, within a fact block: `all w:World, x: w.Class | isSeq[x.(w.association)].` In natural language, this expression is translated as “*for all worlds w, the individuals that every instance x of Class is connected through association, compose a sequence*”. The expression implies that an orderly assignment of numbers to the instances of an association, starting from zero and growing one by one, without skipping numbers. We complete the mapping of ordered relations by adding an expression to forbid the relation between two individuals more than once through the same association.

Finally, the mapping of meronyms, characterizations and mediations is not limited to the ternary field declaration. For each association stereotyped as ComponentOf, MemberOf, SubQuantityOf, SubCollectionOf, Mediation and Characterization, an

additional fact block is generated, which contains the expression: `all w: World | acyclic[w.association,w.Source]`. This fact defines these associations as acyclic, implying anti-reflexivity (e.g. an individual cannot compose itself) and anti-symmetry (e.g. if ‘a’ composes ‘b’, ‘b’ does not compose ‘a’), as prescribed in UFO.

Table 2. Mapping of OntoUML Associations in Alloy.

OntoUML	Alloy
Association	<pre>abstract sig World { association: set Source -> Target }</pre>
Material (connected to a derivation)	<pre>abstract sig World { material: set Source -> Relator -> Target }</pre>
Derivation	<pre>fact derivation { all w: World, x: w.Source, y: w.Target, r: w.Relator x -> r -> y in w.material iff x in r.(w.mediation1) and y in r.(w.mediation2) }</pre>
Ordered Association (<i>isOrdered=true</i>)	<pre>abstract sig World { association: set Source set -> set Int set -> set Target } fact ordering { all w:World, x: w.Class isSeq[x.(w.association)] all w:World, x: w.Class, y:w.OrderedClass lone x.(w.association).y }</pre>

3.2.4.3 Properties: Association Ends and Attributes

The OntoUML metamodel specifies the Property construct for association ends and attributes because they share a great deal of meta-characteristics, like multiplicity and immutability (formalized by the *isReadOnly* meta-attribute). Thus, we define a general mapping for properties, making distinctions for ends and attributes only when necessary.

As we discussed in the previous section, associations give rise to fields within the world signature. The transformation of attributes specified within classes is quite similar. We create a ternary field within the “World” signature for unordered attributes and a 4-ary

for ordered ones. The difference is that instead of specifying using the related elements in the specification, we use the owner class and the attribute's type. Table 3 details this mapping.

Going back to properties in general, we map them into Alloy functions that take, as parameters, an instance of the owner class and a particular world, and return individuals of the property's type. A function generated from an attribute would have a header like `fun property [x: World.Class, w: World] : set DataType`. The function's body accesses the individuals connected to 'x' through a particular attribute or association. Its specification varies according to the property type (attribute or association). Besides, source association ends are defined differently if they are in the source or in the target of the association. Finally, their definition also depends on whether a ternary or 4-ary field formalizes the association. Once more, Table 3 details all these possibilities.

The definition of property multiplicity depends on its value. For properties defined using the common multiplicity values, i.e., zero-one, exactly one, zero-many or one-many, we embed the restriction in the respective association or attribute mapping within the world signature. For example, we map the association "fatherOf", between a father and his offspring as `parentOf: set Father one -> some Offspring`, because a father has at least one child, who has exactly one father.

Whenever custom multiplicities are specified, we enforce them through the specification of an additional fact, as in `all w:World, x:w.Owner | #property[x,w]>=lower and #property[x,w]<=upper`. To define the custom multiplicity, we use the former defined functions.

Another meta-characteristic a property has is entitled `readOnly`. This feature is used to capture specific and existential dependency in OntoUML models (e.g. that is why every mediation must be have `readOnly=true` in the end connected to the mediated type). If a property is defined as `readOnly`, a following fact is generated `immutable_target[Source,association]`. This example refers to an end that is the source of the association. If it were the target, "Source" would be replaced for Target and if it were an attribute, the substitution would be for the owner Class. As we

previously showed, the immutable predicates are defined in the ontological properties module (for more details, refer back to Section 3.2.2).

The last association properties covered by our transformation are the subsetted and redefined properties. For this mapping we adopt the formal subsetting and redefinition semantics defined in (COSTAL; GÓMEZ; GUIZZARDI, 2011). The semantics of subsetting is that a property p_1 subsets a property p_2 if, for every x that instantiates the owner of p_1 , the set of individuals x is related through p_1 is a subset of the set of individuals x is related through p_2 . The semantics of redefinition is similar, but instead of an inclusion constraint, we have an equality one, i.e., the individuals related through p_1 and p_2 are the same. Assuming these definitions, we map p_1 subsetting p_2 as: `all w:World, x:w.Owner | p1[x,w] in p2[x,w]`. Furthermore, we map p_1 redefining p_2 as: `all w:World, x:w.Owner | p1[x,w]=p2[x,w]`.

Table 3 summarizes all mappings related to association ends:

Table 3. Mapping of attributes and association ends into Alloy

OntoUML	Alloy
Attribute (within a class)	<pre>abstract sig World { -- unordered attribute attr1: set Class set -> Datatype -- ordered attribute attr2: set Class set -> set Int set -> set Datatype }</pre>
Property	<pre>-- Attribute fun property [x: World.Class,w: World] : set DataType { x.(w.attribute) } --Source AE fun property [x: World.Opposite,w: World] : set World.Type { (w.association).x } --Target AE fun property [x: World.Opposite,w: World] : set World.Type { x.(w.association) } --Source AE - Material or Ordered fun property [x: World.Opposite,w: World] : set World.Type { (select13[w.association]).x }</pre>

	<pre>--Target AE - Material or Ordered fun property [x: World.Opposite,w: World] : set World.Type { x.(select13[w.association]) }</pre>
Default Multiplicity	<pre>-- 0..1 = lone; 1..1 = one; 1..* = some; 0..* = set abstract sig World { attribute: set Class set -> some Datatype association: set Source lone -> one Target }</pre>
Custom Multiplicity	<pre>-- Attribute (Class: class that owns the attribute) fact { all w:World, x:w.Class #property[x,w]>=lower and #property[x,w]<=upper } --AE (Opposite: type of the end opposite to the property) fact { all w:World, x:w.Opposite #property[x,w]>=lower and #property[x,w]<=upper }</pre>
property1 subsets property2 (subsetProperty)	<pre>-- Opposite: type of the end opposite to the property fact subsetting { all w:World, x:w.Opposite property1[x,w] in property2[x,w] }</pre>
property1 redefines property2 (redefinedProperty)	<pre>-- Opposite: type of the end opposite to the property fact redefinition { all w:World, x:w.Opposite property1[x,w]= property2[x,w] }</pre>
isReadOnly	<pre>fact associationProperties { immutable_target[Source,association] -- Source AE immutable_source[Target,association] -- Target AE immutable_target[Class,attribute] -- Attribute }</pre>

3.2.4.4 Meronymic Properties

Meronymic is a very particular type of relation. As we discussed in Chapter 2, the foundational ontology imposes many constraints on them, like anti-reflexivity, anti-symmetric, weak supplementation, and so on. Furthermore, OntoUML prescribes a set of meta-properties exclusively regarding Mereology, which modelers can customize, as the *isEssential* and *isExtensional* meta-properties of part-whole relations. In this

section, we describe the refactored mapping of these meronymic constraints and meta-properties.

We start with the mapping of the weak supplementation axiom, which traces back to UFO. The axiom states that every whole must contain at least two proper parts, because a whole contains a single part, the whole and the part are the same individual. In order to forbid ontological inconsistent models at the syntactical level, OntoUML contains an embedded constraint regarding whole specification: the sum of the lower bound cardinalities on the part ends of the meronomies directly or indirectly connected to a whole must be greater or equal to two. Although very restrictive, it is not enough to forbid every inadmissible model instantiation. Wholes composed by parts that have a common intersection might bypass this constraint. In order to cope with that and enforce a minimum of two parts, even if the cardinality constraints are not properly defined, we add a fact block for each meronymic, containing the following expression:

`all w: World, x: w.Whole | #(part_1[x,w] + ... + part_n[x,w])>=2.` This constraint is not concerned with the actual defined cardinality defined for the relations, but only in enforcing the two minimum parts.

A meronymic relation, as defined in the foundational ontology, must always be anti-reflexive, anti-symmetric and acyclic. This means that no individual can be part of itself, that if ‘a’ is part of ‘b’, then ‘b’ is not part of ‘a’, and that an individual cannot be an indirect part of itself, respectively. Since acyclic implies anti-reflexivity and anti-symmetry, we only include the following expression: `all w: World | acyclic[w.meronymic, w.Whole]`, which meronymic relations as acyclic.

Now we discuss meronymic meta-attributes and their respective mappings. We start discussing *isEssential* and *isImmutablePart*. If a modeler defines a part-whole relation as essential, it means that the instances of the whole are existentially dependent on the instances of the part. Simply put, throughout the existence of the whole, its parts may never change. Immutable parts are a weaker version of essential parts. Instead of an existential dependency, they characterize a specific dependency. Therefore, the constraint would read, “*While instantiating a whole type, the same parts must compose an individual*”. When set to true, we enforce both meta-attributes by making a predicate call, as in: `immutable_target[Whole,meronymic]`.

Opposed to the previous constraints, there is *isInseparable* and *isImmutableWhole*. They also capture existential and specific dependencies, but this time, from the part to the whole. If a part composes an inseparable whole, it must always do so while it exists. If a part composes an immutable whole, it means that whilst instantiating the part type, it composes the same wholes. We enforce these meta-attributes through a single predicate call: `immutable_source[Part,meronymic]`.

In the following, we map the *isShareable* meta-attribute. One of the solutions to solve the semantic gap identified in UML, regarding the difference between Aggregation and Composition (GUZZARDI, 2005, chap. 8), *isShareable* is a multiplicity constraint. When set to true for an association, it means that a part can compose at most one instance of the respective whole and when it does, it cannot compose any other whole. We achieve that in Alloy by creating a fact block that contains two expressions:

- `all w:World, x : w.Part | lone whole[x,w]`, which enforces the composition of at most one whole; and
- `all w:World, x : w.Part | some whole[x,w] implies no (whole_1[x,w] + ... + whole_n[x,w])`, which forbids the composition of wholes of any other type.

Lastly, we present the mapping for the *isExtensional* meta-property, which can only be set on collectives. This property is redundant, because it means that every part-whole relation connected to the collective has *readOnly* set to true in the part end. For the sake of correctness, even if the modeler does not set all relations as read only, we generate the expressions as if she had done it.

Table 4 lists the summary of all mappings regarding meronymic constraints and meta-properties.

Table 4. Mapping of meronymic constraints and meta-properties.

OntoUML	Alloy
Weak Supplementation Axiom	<pre>fact weakSupplementationConstraint { all w: World, x: w.Whole #(part_1[x,w]+ ... + part_n[x,w])>=2 }</pre>
Acyclic Meronyms	<pre>fact acyclic { all w: World acyclic[w.meronymic,w.Whole] }</pre>

	<pre>-- ordered association fact acyclic { all w: World acyclic[select13[w.meronymic],w.Whole] }</pre>
<i>isEssential</i> and <i>isImmutablePart</i>	<pre>fact associationProperties { immutable_target[Whole,meronymic] }</pre>
<i>isInseparable</i> and <i>isImmutableWhole</i>	<pre>fact associationProperties { immutable_source[Part,meronymic] }</pre>
<i>isShareable</i>	<pre>fact nonShareable { all w:World, x : w.Part lone whole[x,w] all w:World, x : w.Part some whole[x,w] implies no (whole_1[x,w] + ... + whole_n[x,w]) }</pre>
<i>isExtensional</i>	<pre>fact associationProperties { immutable_target[Whole,meronymic1] immutable_target[Whole,meronymic2] ... }</pre>

3.2.4.5 Relator and Mode Constraints

Simply put, relators are objectification of material relations. As we discussed in Chapter 2, UFO assigns an axiom upon relators that requires every instance to mediated at least two disjoint entities. The reflection of this axiom on OntoUML is a syntactical constraint that requires the sum of the lower bound cardinality of all mediations, directly or indirectly connected to a relator, to be greater or equal to two. This constraint suffers from the same problem of weak supplementation: it has a “loophole”. Whenever all mediated types can be simultaneously instantiated by the same individual. To enforce the desired ontological constraint, regardless of the syntactical rule, we create fact blocks containing the following expression: **all** w: World, x: w.Relator | **#(mediated_1[x,w]+ ... + mediated_n[x,w])>=2**. This rule specifies that every relator instance mediate at least two distinct individuals, regardless of how many mediations are connected to the relator.

Ontologically, mediation associations are internal relations used to connect object types to relators, with the particular property of characterizing an existential dependency from the relator to the mediated type. Syntactically, this means that the

target end of every mediation (the end connected to the mediated type) must be set as `readOnly=true`. The existential dependency captured by a mediation is mapped as a predicate call, as in `immutable_target[Relator,mediation]`.

Characterizations are very similar to mediations, but instead of specifying relational properties, it formalizes the relation between an intrinsic property (either a mode or a quality) and the individual it characterizes, the property's bearer. A property can characterize other intrinsic properties, objects or even relators. An object, however, must always be the “ultimate” bearer. For example, if James has a 38° fever, we interpret that the temperature is a quality that inheres in the James’ fever, which is a mode that inheres in James, and object. OntoUML has two syntactical rules to enforce these ontological notions:

- every intrinsic property must be directly connected to a unique characterization relation; and
- every mode must be directly or indirectly characterize an object class or a relator.

Just like in the relator rule and the weak supplementation, the syntax is not enough to enforce the complete ontological definition. Particularly, for recursive characterization definitions, intrinsic properties can indirectly characterize themselves. To forbid that, we include a single fact that forbid cycles in the combined extension of all characterizations: `all w: World | acyclic [w.charac1 + w.charac2, w.Mode1 + Quality2]`. The expression calls the acyclic predicate, defined in the util/relation module, passing as parameters the union of all characterizations and the union of all intrinsic properties.

Finally, as characterizations impose existential dependencies (the property depends on the characterized individual), each generates a call to the immutable predicate, just like mediations. Table 5 summarizes the mappings involving relators, mediations, modes, qualities and characterizations.

Table 5. Mapping of relators, mediations, modes, qualities and characterizations.

OntoUML	Alloy
Existential dependency	<pre>fact associationProperties { immutable_target[Relator,mediation] immutable_target[Mode,characterization]</pre>

	<pre> immutable_target[Quality,characterization] }</pre>
Acyclic Characterization	<pre>fact acyclic { all w: World acyclic[w.characterization1+w.characterization2,w.Model+Model2] }</pre>
Relators Rule	<pre>fact relatorConstraint { all w: World, x: w.Relator #(mediated_1[x,w]+ ... + mediated_n[x,w])>=2 }</pre>

3.2.4.6 Datatypes

As classes, datatypes characterize a set of individuals that share common characteristics. Thus, they can participate in associations, own attributes and be specialized into other datatypes. Unlike classes, instances of datatypes do not exist in time, and their values define their identities. For example, if two persons weigh 80 kg, the same instance of the weight datatype (a point in the quality dimension) characterizes both persons.

The particular features of datatypes impose a transformation that follows a completely different path than the one for class. To start, since datatypes are atemporal, we cannot declare them as fields of the world signature. Thus, we represent them as individual signatures that subset the general datatype signature, e.g. `sig CustomDatatype in DataType {}.`

The reason we declare a general datatype signature and make every custom datatype be included in it, is because Alloy only allows multiple inheritance if all parents are overlapping *a priori* (JACKSON, 2012). This design decision, though, requires the specification of additional constraints. First, we must declare that every instance of the general datatype is an instance of the ones declared in the model. We do that in an additional fact block, through the expression `DataType = Datatype1 + ... + Datatype2.` To compete the specification, we add a disjointness constraint, for datatypes that: (i) are not specializations of one another; (ii) do not share a common super-type; and (iii) do not share a common subtype. This constraint is enforced through the expression `disj [Datatype1, (Datatype-2+Datatype-n)].`

By changing the way we specify datatypes, we invalidate the mapping of attributes and associations involving exclusively datatypes. The mapping of attributes changes from fields of the world signature to fields of the owner type, as in `attribute: one Int`.

OntoUML does not prescribe any stereotype for defining associations between datatypes. So whenever one that relates two datatypes is inputted, the transformation ignores the stereotype and generates a field declaration within the datatype connected to the source of the association, as in `association: one CustomDatatype`. We map multiplicities analogously to the way we map multiplicities for associations between classes, but this time, without projecting on the world signature, as in: `all x: CustomDatatype | #x.association>=lower and #x.association<=upper`.

Table 6. Mapping of datatypes and related elements and constraints in Alloy

OntoUML	Alloy
Datatype	<code>sig CustomDatatype in DataType {}</code>
Attribute (within a datatype)	<code>sig CustomDatatype in DataType { attribute: one Int }</code>
Association (between datatypes)	<pre>-- ignores stereotypes sig CustomDatatype1 in DataType { association: one CustomDatatype2 } fact multiplicity { all x: CustomDatatype1 #x.association>=lower and #x.association<=upper all x: CustomDatatype2 #association.x>=lower and #association.x<=upper }</pre>
Datatype identity	<code>fact datatype_identity{ all x,y : LeafDatatype x.attr1=y.attr1 and x.attr2=y.attr2 implies x=y }</code>
Additional facts	<code>fact additionalDatatypeFacts { DataType = Datatype1 + ... + Datatype2 disj [Datatype1, ..., Datatype-n] }</code>
Enumeration	<code>enum Enumeration {Literal1, Literal2, ..., Literal-n}</code>

As we discussed in the beginning of this section, the identity of a datatype regard its properties values. To enforce that behavior, we add the following expression: `all x,y : CustomDatatype | x.attr1=y.attr1 and x.assoc=y.assoc implies x=y`. Note that this constraint includes the comparison of all attributes and association owned by a

datatype or inherited by it. Furthermore, we only generate such constraints for the leaf data-types, i.e. the ones that have no subtypes.

Lastly, the mapping of Enumerations, a particular type of Datatype that correspond to a discrete sequence of literals, is straightforward: `enum Enumeration {Literal1, Literal2, Literal-n}`. In Alloy, this declaration is a syntactical shortcut, equivalent to declaring `sig Enumeration {}` followed by `one sig Literal1, Literal2, Literal-n extends Enumeration {}`. Differently from regular datatypes, enumerations do not support specialization, attributes or event relation between themselves.

3.2.4.7 Primitive Types: Strings, Integers and Boolean

OntoUML lacks a basic type library containing the most common types of data used, like string, char, natural, integer, floating point, Boolean, etc. Even so, we propose a few mappings of primitive types like these, in order to take advantage of Alloy's basic types. To do that, we use the datatype's name. If a type is named String (or string), instead of creating new datatype signature, it is mapped to Alloy's embedded String. For integers, the same thing happens, a mapping to the primitive Integer signature. Lastly, we map Boolean types to a Bool signature, defined in a standard library.

By mapping types to the primitive String signature instead of an ordinary signature, we provide modelers with the ability (and the obligation) to define a String range, i.e., the values a string may assume. This is required because the analyzer is not able to generate string values automatically. This mapping has a cognitive advantage, since the simulation will show actual strings, improving the readability of the generated worlds. Furthermore, it is compliant with datatype identity, since two strings with the same values are the same. This mapping, however, due to Alloy's limitation, does not allow user to use complex string operations, only value assignments and comparisons are available.

By mapping integer attributes to the primitive Int signature, we also increase the readability of the simulation results. In this case, Alloys support is more refined, so in addition to the ability of assigning values and comparing them, users are able to calculate sums of a set of integers, besides the basics arithmetic operations, like

addition, subtraction, multiplication, division and modulus. Alloy bounds the instantiation of the Int signature in a particular way. For each command, one can define the size of the integer set, like in `run predicate for 5 but 7 int`. This command generates $2^7 = 128$ values, ranging from -64 to 63.

Alloy provides no native support for storing primitive Boolean values in signature fields. To cope with this limitation, the designers developed an Alloy module as a work around, defining a signature named Bool, which is specialized in two singletons: True and False. Alloy provides support for Boolean operations through custom functions and predicates.

Table 7. Mapping of String, Boolean and Integer.

OntoUML	Alloy
String	<code>String</code>
String Possibilities	<code>all s: String s="String1" or s="String2" or ... or s="String-n"</code>
Boolean, Bool	<code>util/Bool</code>
Integer, Int	<code>Int</code>
Int range	<code>-- 7 int means $2^7 = 128$ values, from -64 to 63</code> <code>run predicate for 5 but 7 int</code>

3.2.4.8 Generalizations and Generalization Sets

In OntoUML, the generalization construct defines an inheritance relation between two classes. The class connected to the origin of the relation is the “child” class, and the other the “parent”. By defining a generalization, one states that the extension of the child class is included in the extension of the parent class. Furthermore, one asserts that the child inherits every property defined for the parent. In Alloy, we represent generalizations independently from the specification of the related classes, and since it does not make sense to talk about property inheritance, we only specify the extension inclusion, as in: `Child in Parent`.

A Generalization Set (GS) is a group of Generalizations of a common supertype. It captures a common criterion of specialization for all subtypes and restricts their

instantiation. A specialization criterion is involved in the definition of why an instance of a type becomes an instance of one of its subtypes. To clarify, consider the types Person, Man, Woman, Child and Adult. A person is classified as Male or Female according to one's gender, whilst is classified as Child or Adult through the evaluation of one's age.

Besides the group of generalizations, a GS has two Boolean meta-properties: *isDisjoint* and *isCovering*. The former regards the multiple instantiation of the respective child classes: if set to true, no individual can instantiate more than one subtype at the same time, if set to false, no restriction exists. Whenever set to true, the transformation generates the following expression: `disj[Subtype1, ..., Subtype-n]`. The *isCovering* meta-property, if set to true, requires every instance of the common supertype to be an instance of at least one of the subtypes in the generalization set. To guarantee that, the transformation maps a complete GS to the following expression: `Parent = Subtype1 + ... + Subtype-n`.

Table 8 summarizes all mappings regarding generalizations and GS's.

Table 8. Mapping of generalizations and generalization sets.

OntoUML	Alloy
Generalization	<code>fact generalization { Child in Parent }</code>
Generalization Set	<code>fact generalization_set{ disj[Subtype1, ..., Subtype-n] -- isDisjoint=true Parent = Subtype1 + ... + Subtype-n -- isCovering=true }</code>

3.2.5 Transformation Parameters

To cope with challenges of simulating partial OntoUML specifications and to provide modelers with flexibility during the validation, we propose the usage of transformation parameters. We discuss each of them in the following sub-sections.

3.2.5.1 Identity

UFO, the foundational ontology that molds OntoUML, has an axiom that requires every object to follow an identity principle. This principle is a sort of function that allows us to differentiate individuals through space and time. According to the foundational ontology, ultimate sortal types (in OntoUML they are classes stereotyped as kind, quantity and collective) provide the individuation criteria for their instances. This axiom gives rise to several syntactical constraints in OntoUML. For example, no instance of an ultimate sortal can be a direct or indirect subtype of another ultimate sortal. Another example is that every subkind, role and phase must be a direct or indirect subtype of an ultimate sortal. A natural consequence of requiring a unique identity for individual is that all classes that provided identity are disjoint from one another, i.e., no individual can ever instantiate more than one ultimate sortal. In Alloy, we enforce this rule through the following expression:

```
disj[Kind1,Kind2,Collective1,Quantity1]
```

A partial model, however, may have some classes with an undefined identity principle. Structurally, that means that either a subkind, role or phase does not specialize an ultimate sortal, directly or indirectly, or a category, roleMixin or mixin does not generalize sortal classes. The former case occurs on model fragments formalizing only a subset of a class hierarchy, whilst the latter occurs on generic models that only define non-sortal classes.

To cope with the undefined identities, we propose a parameterization that allows users to choose between enforcing the identity axiom or not. If enforced, the Alloy specification will state that every object is an instance of one ultimate sortal. We do that by making the set of all objects that exists in world to be equal to the union of ultimate sortals that exist in that world, like in the following code:

```
all w:World | w.exists:>Object =  
(w.Kind1+w.Kind2+w.Collective1+...+w.Quantity1)
```

If not enforced, the transformation will replace the previous Alloy expression with the one that enforces every object individual to be an instance of at least one object class. To do that, we make the projection of the Object signature on the things that exist in a world to be equal to the union of all object class extensions, like in the following code:

```
all w:World | w.exists:>Object =
(w.Kind1+w.SubKind1+w.Collective1+...+w.Role1+w.Category1)
```

Furthermore, note that, if two classes have no identity, we cannot say for sure that they are disjoint. A modeler might be simulating two subkinds that specialize the same kind, making them overlapping, or that specialize two different kinds, making them disjoint. In the same way, if the modeler were simulating multiple mixins, they would be overlapping if they generalize the same sortals. These mixins, however, would be disjoint if they generalize completely independent sortals.

To cope with the aforementioned lack of information, we envision three possibilities. The first is to be conservative and state that every two distinct identity-less classes are disjoint if they are not direct or indirect subtypes of one another and they do not share a third class as a common subtype or supertype. The second is to be more liberal and state that if we cannot conclude that two classes are disjoint, then they might be overlapping. Lastly, we can refrain from making any pre-defined decision and interact with the modeler to decide which classes are overlapping and which are not.

To exemplify these three alternatives to deal with identity issues in a partial model, consider the fragment depicted in Figure 7. The conservative approach would conclude that “Colored Object” and “Spherical Object” are overlapping, since these classes have a common super-type: “Physical Object”. It would also conclude that “Male” and “Child” are overlapping, since they have a common subtype: the subkind “Boy”. However, it would consider “Moving Object” and “Physical Object” as disjoint. The same decision would apply to “Male” and “Moving Object”, and so on. The liberal approach assumes that “everything is possible”. Therefore, there might be an instance of “Male”, which is also an instance of “Moving Object” and “Physical Object”. In the last alternative, the user would have to indicate for every possible combination of two top-level classes (classes without parents) if they are disjoint or not. For instance, the user might decide that “Moving Object” possibly overlaps with “Physical Object”, but they do not overlap with neither “Male” nor “Child”.

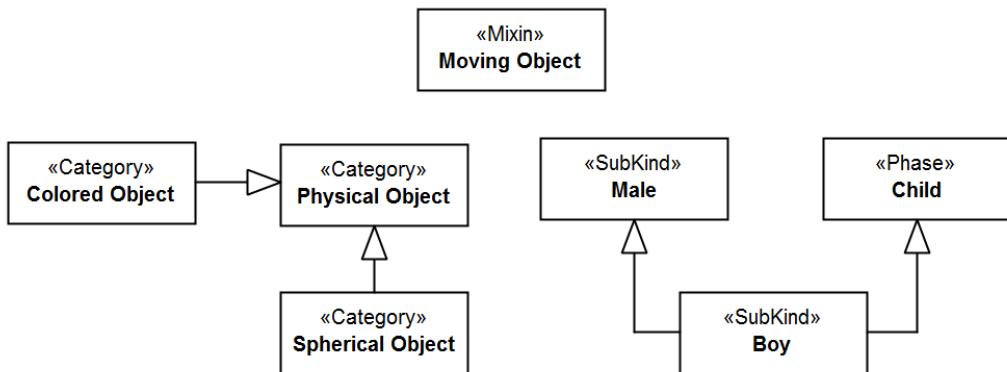


Figure 7. Partial OntoUML diagram with identity issues in all classes.

If modeler decides for the conservative approach, the transformation will generate an expression for each top-level class, enforcing its disjointness with the union of every other top-level class. The following expression exemplifies the expression required for the category “Physical Object” in Figure 7.

```
disj [PhysicalObject, (Male+Child+MovingObject)]
```

Note that if some classes have identities and some do not, enforcing identity will limit the analyzer to generate branches in which the extension of identity-less classes are always empty. If no class has identity, enforcing the axiom causes the analyzer to generated only empty worlds.

3.2.5.2 Weak Supplementation

UFO’s part-whole theory prescribes that every whole contains at least two proper parts – the Weak Supplementation Axiom. This axiom’s reflection on the OntoUML syntax requires that the sum of the lower bound cardinalities of all meronymic relations connected to a whole must be equal or greater than 2. Although this syntactical rule restrict many undesired cases, it is not enough. In cases where the same individual can simultaneously instantiate multiple types that compose the same whole type, the model allows instantiations that break the weak supplementation axiom. To forbid these cases, we include one fact per whole in the alloy specifications, which constraints the minimum of parts in every whole individual:

```
all w:World, x:w.Whole | #(x.part1[w]+x.part2[w])>=2
```

Every model that contains wholes connected to part-whole relations whose sum of their lower bound cardinalities, is less than two, automatically fails the weak supplementation axiom. We are not concerned here with all the possibilities one can fail the weak supplementation, but only a single case: wholes connected to exactly one meronymic relation whose cardinality in the part end is exactly one. We are only concerned with this case because the aforementioned Alloy fact will preclude the analyzer from generating any instances of the whole type. Moreover, we are only concerned with it because we learned that modelers might desire to analyze the behavior of a single part, or even the dynamics of an abstract class that really only has one part.

To cope with the aforementioned problem, we provide users with a Boolean transformation parameter: the ability to enforce (or not) the weak supplementation axiom. If enforced, we include the aforementioned fact in the product of the transformation; otherwise, we just ignore it.

3.2.5.3 Relator's Rule

The problem we address here is analogous to the weak supplementation one, described in the previous section. That is because relators are required to mediate at least two distinct individuals, in the same way that wholes are required to be composed at least two parts. The reflection on OntoUML's syntax is that, for every relator, the sum of the lower bound cardinalities of all mediations directly or indirectly connected to it must be greater or equal to two. Again, just like for wholes, this constraint is “breakable” if the same individual can instantiate all types mediated by a relator. Our solution in the alloy code is analogous:

```
all w:World, x:w.Relator | #(x.mediated1[w]+x.mediated2[w])>=2
```

Unlike part-whole relations, mediations do not accept optional cardinality. Therefore, a relator fails the constraint in only two situations. First, if the relator has no direct/indirect mediation. Second, if it has exactly one mediation, with a cardinality of exactly one in the mediated end. Notice that if the aforementioned fact is included for relators that fail the constraint in the second case, the analyzer will not be able to generate worlds in which the relator has a non-empty extension.

To simulate a model containing a relator with the aforementioned problem successfully, one needs to relax the relator constraint. The transformation does that by ignoring the aforementioned cardinality Alloy expression.

3.2.5.4 Anti-rigidity

A class is anti-rigid if, and only if, its instantiation is a contingent situation, i.e., if any given individual instantiate an anti-rigid class in a world w , there is at least one possible world in which the same individual exists and does not instantiate the class. Notice that anti-rigidity is a matter of possibility, not of necessity. For example, consider the phases a person goes through life: Child, Adult and Elder. They are all anti-rigid classes, since a person's age characterize them, which increases over time. Every person is born as a child, and if she lives until her 60's, she goes through all the aforementioned phases. Now, if due to an unfortunate event, a child passes away, it will have only instantiated the first phase.

To generate simulations that encompass all life cycle possibilities, we chose not to enforce anti-rigidity by default. Instead, we propose it as a transformation parameter. If one desires to inspect branches in which every individual "lives" enough to instantiate anti-rigid classes and cease to do, the following additional constraint is included in the Alloy specification:

```
all x:World.AntiRigidClass | some disj w1,w2:World | x in w1.exists and x
in w1.class and x in w2.exists and x not in w2.class
```

3.3 COMPARISON TO PREVIOUS APPROACHES

In this section, we compare our transformation from OntoUML to Alloy with the ones proposed by Benevides (2010b) and Braga (2010). We do that through four categories: Meta-model, World Structure, Coverage and Mapping. Throughout the comparison, we point the things we kept from the previous proposals and the things we improved upon.

3.3.1 Meta-model

Benevides' and Braga's transformations are defined using the same meta-model, published in (BENEVIDES, 2010). As we already discussed in the motivation for revisiting these transformation, the OntoUML meta-model has been redesigned in (CARRARETTO, 2010), when it was baptized as Reference OntoUML (RefOntoUML), and updated in (ALBUQUERQUE; GUIZZARDI, 2013). Our transformation is compatible with RefOntoUML¹, available on the OntoUML Lightweight Editor's (OLED) website, maintained by the Ontology and Conceptual Modeling Research Group (NEMO).

3.3.2 World Structure

In our transformation, we adopt the world structure proposed by Benevides: a Kripke structure, containing past, future, current and counterfactual worlds, which we already discussed in the definition of the world module (see Section 3.2.1). Along with the structure, we kept two constraints proposed by Benevides. One that enforces continuous existence of individuals, i.e., states that if an individual exists in a world and ceases to exist in an immediate future, it cannot exist again in all future worlds. The other constraint forbids the appearance of empty worlds, i.e. worlds in which no individual exist. Although these constraints do not follow from UFO, they are useful to approximate the simulation to our common sense of existence through time.

Braga's approach represents the admissible instances of the ontology using the "State" concept, instead of World. On one hand, they are very similar, because they are both representations of a possible state of affairs, and thus, represent how individuals can exist and relate to one another in a given moment of time. They are not so alike, on the other hand, because states are not part of a more complex structure. They are just a linear ordered sequence of snapshots. Comparing the world and the state structures to well know data structures, we say that the former resembles an n-ary tree, whilst the latter a finite total order.

¹ The latest version of the RefOntoUML metamodel is available at: <https://code.google.com/p/ontouml-lightweight-editor/>

The reason we kept Benevides' World structure, instead of Braga's State structure, is that in a more complex structure, we can perform a more interesting analysis. Furthermore, if desired by the user, the world structure can generate a linear sequence of worlds. We discuss the generation of pre-defined world structures in the next section.

Lastly, we kept from Braga's proposal the Alloy function entitled "visible". As we said before, it does not affect the simulation, but facilitates the user interface customization when inspecting the simulation results.

3.3.3 Coverage

Both previous approaches covered most of the ontological distinctions made by OntoUML, like rigidity, identity and existential dependency. They fall short, however, in mapping elements and meta-attributes inherited from UML that impose refined logical constraints and that are very useful in practice.

Firstly, the previous approaches abstain from the explicit representation of **association ends**. Even though they do not contribute to the semantics generated specification, by defining them as Alloy functions, we improve the usability of the generated code and modularize the generation of other constraints, like multiplicity and weak supplementation, improving the readability of the generated code. Furthermore, the mapping of the association ends is required for the transformation to be compatible with the OCL mapping described in (GUERSON; ALMEIDA; GUIZZARDI, 2014).

Both approaches also do not provide support for property **subsetting** and **redefinition**, and neither for **association specialization**. These simple mappings are very useful for modeling and often required, as we latter show in the Relation Specialization anti-pattern (see Section 5.17). We enforce all of them through the generation of constraints, following the semantics defined in (COSTAL; GÓMEZ; GUIZZARDI, 2011).

Furthermore, both previous approaches neglect the meta-property named ***isOrdered***, owned by association ends and inherited from UML. We provide support for it by orderly assigning a natural numbers to each relation.

The previous transformations also did not assign mappings for two meta-properties regarding part-whole theory, namely ***isExtensional***, owned by collectives, and ***isShareable***, owned by the four types of part-whole relations. We are aware that both features are redundant, i.e., imply the specification of particular values for other meta-properties. *IsExtensional* means that every part-whole relation connected to the whole has *isEssential* set to true, whilst *isShareable* imply a maximum cardinality of one, on the whole end. We argue, however, that one cannot expect modelers to set all of these meta-properties, and thus, we map whatever a model defines.

We also build upon the previous approaches, by providing special mappings for some very common primitive types of data, namely **Integer**, **String** and **Boolean**. By doing that, instead of mapping all data-types as additional signatures, we provide users with more resources to validate their models, since they are able to specify expressions using functions and predicates specially designed for these data types. In addition, we aim at diminishing the cognitive load of understanding the simulation results, assuming that one understands values like “John”, 1 or True, easier than “String0”, “Integer0” or “Boolean0”.

Lastly, we define a basic support for the **Quality** meta-class, which was only added in the meta-model in Albuquerque’s update (ALBUQUERQUE; GUIZZARDI, 2013). As we previously shown, we only make no distinction between nominal, measurable and non-measurable qualities.

Overall, our transformation covers a wider specter of the language, mainly regarding logical meta-properties and elements specified only in the more recent versions of the OntoUML meta-model.

3.3.4 Mapping

In this section, we discuss the changes we made on mappings already defined by the previous approaches. In each case, we provide explanation for the reasons the change was required, as well as its associated benefits.

The most impacting alteration we made regard **class** mapping. We map all classes as fields within the world signature, whilst the previous approaches only assigned such mapping for anti-rigid and semi-rigid classes. In addition, we separate the class mapping from the **generalization** mapping. These combined decisions allow the transformation to cope with partial models (e.g. subkinds without parents) and multiple inheritance, unfeasible in the previous approaches.

We also take a different approach regarding rigidity. By mapping all classes as fields, we do not commit to rigidity, anti-rigidity or anti-riidity. We enforce them through additional facts. We also argue that the way we deal with anti-riidity is more useful than the previous approaches. Benevides always enforces anti-riidity, whilst Braga never do so. Since both approaches are useful in different situations, we added a parameter to the transformation, providing the user the decision at run time.

The previous approaches map **categories** as Alloy functions, unlike the other type of classes. We assume that this decision was motivated to obtain better performance on the analyzer. The problem is that it turns out to be the source of many representation problems. First of all, the mapping does not handle categories that are not ultimately specialized in sortal types, a common situation for partial models or core ontologies, like UFO-S (NARDI et al., 2013). Furthermore, it precludes the definition of role mixins as subtypes of categories, a very common modeling construction.

Braga and Benevides propose the representation of relations containing an **existential dependency** (*readOnly=true* on at least one of the association ends) as fields of the dependent classes signatures. Mainly, we changed that because of the new class mapping, but even if we had not, the original mapping does not account for bidirectional dependency or anti-rigid types as the dependents. We map every **association** as signature fields and enforce the existential dependency through additional facts.

The particular case of **material** associations also needed refactoring. The original proposal was to represent them as ternary fields within the World signature (e.g. {World, Source, Target}). The problem with this mapping is that it precludes multiple relations of the same type between two individuals. To solve this limitation, we propose materials to be mapped into a 4-ary relation, (e.g. {World, Source, Relator, Target}). Note that this mapping is limited to material relations connected to a Relator through a derivation. We still map undefined material associations as ternary relations.

A side effect of changing material association representation is the need to redefine the **derivation** mapping. In our approach, instead of generating an extra field, we map the restriction into a fact. The results, however, are the same.

Regarding part-whole theory, we improve the previous approaches by adding the **Weak Supplementation axiom** mapping for whole types, as well as the explicit enforcement of the common binary properties of part-whole relations: anti-reflexivity, anti-symmetry and acyclicity. We enforce the same properties for **characterizations**. Furthermore, we complement the **relator** mapping by enforcing the minimum of two distinct mediated types.

Lastly, we map **generalizations set** as Braga proposed: a fact if set as disjoint and another if set as complete. We discarded Benevides proposal because it failed to handle multiple orthogonal generalization sets of a common supertype.

3.3.5 General Remarks

In summary, we propose in this chapter a more stable and complete transformation from OntoUML to Alloy. Besides the aforementioned improvements, we generate a more homogeneous specification, which are simpler to understand, since one can see the direct impact of each language construct.

We aim to transform elements individually. For example, categories used to be a function, defined in term of its subtypes. Now every class has an independent transformation, as well as each generalization. The previous transformations often define mappings for modeling patterns, instead of mappings for each element

individually. Whenever one modifies these patterns, the previous transformations could not deal with it. Our design decision provides more stability for the transformation and a more faithful representation in Alloy, since we transform even inconsistencies in their original form.

3.4 SIMULATION SCENARIOS

Our main research goal is to provide a validation framework that requires little learning requirements. The refactored transformation from OntoUML to Alloy described in this chapter is a very promising tool for ontology validation, however it still requires users learn to yet another language. Modelers without any knowledge in Alloy could only use require unconstrained simulations, which is not efficient for validation. In addition, even though Alloy focuses on simplicity, one still needs to be well versed in logics to be able to explore its full potential.

Furthermore, if we just provide a simulation tool, we face users with a dilemma: how should they simulate their models and which properties should they check? To cope with this problem, we propose a number of simulation scenarios, which users can parameterize and combine in order to validate their models. This approach is heavily inspired in (JANSSEN et al., 1999), an original work in which the authors show how model checking techniques can be applied to validate business process models, without users being trained in them. Just like Janssen and his colleagues, we transform the pre-defined scenarios in natural language sentences, which users can easily understand and that have direct mapping to Alloy predicates.

Notice that we identify the parameterization of a scenario in the provided sentence by using brackets ([]). It has two uses: first, to indicate the need to specify a numeric value, like “*at least [n] instances of class*”. Second, it can detail alternative options, like “[*every / no / at least / at most / exactly*] worlds must have”.

One of the tasks users perform when using the simulation tool is to generate a branch and assess whether it is admissible by the ontology or not. That implies having to process cognitively many things, like the generated world structure, the dynamics of object creation and destruction and the dynamics of the instantiation of anti-rigid types.

By the usage of pre-defined scenarios, users partially know what to expect from the simulation, diminishing the cognitive work facilitating one's analysis. For example, if one requires all generated worlds to contain the same individuals, one would not need to keep track of object creation and destruction when moving throughout worlds.

Although the simulation scenarios allow modelers to easily simulate their models and check some properties, they will not identify the source of the problem and point to solutions. For those cases, we propose a catalogue of anti-patterns, which we will present and discuss in Chapter 6.

In the following subsections, we discuss the simulation scenarios, their relations and the kind of properties they are able to check.

3.4.1 Branch Structure Constraints

In the refactored OntoUML2Alloy transformation, we kept the Kripke's World Structure proposed by Benevides. Thus, simulating a model randomly generates a finite world branch, i.e., a finite set of worlds connected through successor and predecessor relations.

In order to help modelers specify or restrict the type of world structure they want to see, we propose a set of branch scenarios.

3.4.1.1 *Linear Branch*

This scenario defines that every simulation will generate a linear world structure, i.e., a branch in which exactly one world does not have a successor and exactly one does not have a predecessor. All the others worlds must have a predecessor and a successor. This structure is how we commonly think of things, a linear sequence of events. Therefore, it relieves users from the cognitive work of understand the world order. Figure 8 depicts an example of linear branch.

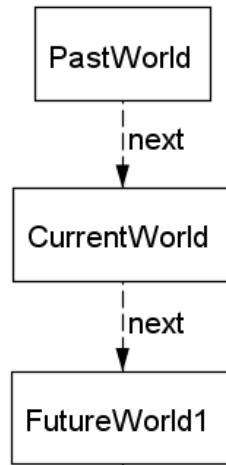


Figure 8. Example of a linear branch.

We present the phrase that characterizes this scenario and its respective Alloy predicate in Table 9.

Table 9. Linear Branch – scenario description.

Sentence
<i>I want to see a linear story.</i>
Alloy Expression
<pre> one w:World no w.next one w:World no next.w </pre>

3.4.1.2 Alternative Futures

This scenario defines a world branch composed by a unique world that leads to alternative futures, as depicted in Figure 9. It does not generate counterfactual or past worlds. Branches fitting this pattern are useful to analyze what can happen to an individual after a given situation. For instance, if a couple is married in a world, some possible futures are: they can either continue to be married, break up or even break up and marry other people.

As we previously discussed, the world structure we adopt classifies worlds according to their position w.r.t. the current world. That means that more than one classification applies to the same structure, depending on where the current world is. In Figure 9, for instance, if we change the current world's position from the starting point to one the

futures, the starting point would change to a past, all other futures would change to counterfactual and no world would be a future.

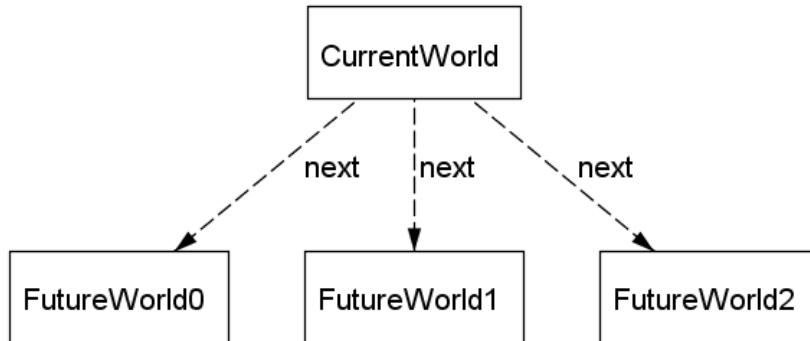


Figure 9. Alternative futures example.

Table 10 presents the natural language description of this scenario (as if by the modeler is demanding it) and the respective Alloy expression that a user should add to the Alloy specification (within either a fact or a predicate).

Table 10. Alternative Futures – scenario description.

Sentence
<i>I want to see different outcomes for a same situation.</i>
Alloy Expression
<pre>-- alternative futures one w:World no next.w && all w2:World w!=w2 implies w2 in w.next</pre>

3.4.1.3 Counterfactual Worlds

Counterfactual worlds exemplify alternative possibilities in the past, i.e., alternative future worlds from a past world. This scenario is specified as a world branch that contains at least two distinct worlds, w_1 and w_2 , which share a common past world and either w_1 and w_2 have a next world, as depicted in Figure 10. This type of scenario is also useful to analyze alternative turn of events.

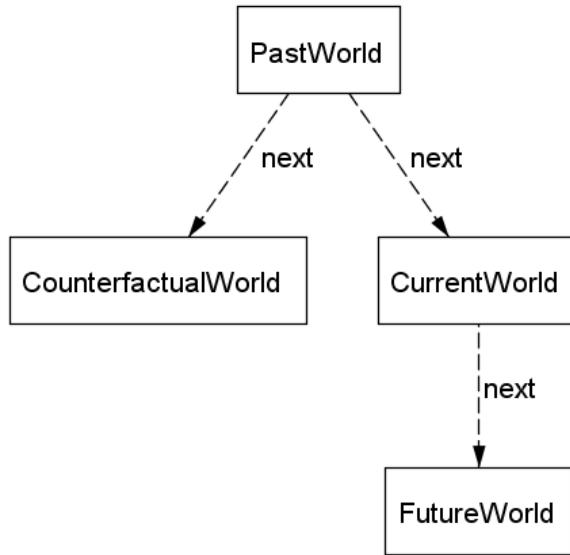


Figure 10. Counterfactual world example.

In Table 11, we present the sentence that express a user's demand for this scenario. Furthermore, we provide the Alloy expression required to generate simulations with the structure previously described.

Table 11. Counterfactual Worlds – scenario description.

Sentence
<i>I want to see that things may have taken a different outcome in the past.</i>
Alloy Expression
<pre>-- counterfactual_worlds some w1,w2:World w1!=w2 && next.w1=next.w2 && (some w1.next or some w2.next)</pre>

3.4.1.4 Branch Depth

The depth of a branch corresponds to the number of consecutive worlds it has, i.e., a set of worlds within a branch that characterize a linear branch by themselves. Figure 10, for instance, has a minimum depth of two and a maximum depth of three, (PastWorld -> CounterfactualWorld; PastWorld, CurrentWorld and FutureWorld). Notice that the Alternative Futures scenario implies an exact world depth of two, since all futures come directly from the same world.

Table 12 provides the representation of this scenario in natural language, alongside with the Alloy expression that characterizes it.

Table 12. Branch Depth – scenario description.

Sentence
<i>I want to see a story composed [at least / at most / exactly] of [n] consecutive worlds.</i>
Alloy Expression
<pre>-- minimum world depth some w1, w2, w3:World w2 in w1.next and w3 in w2.next -- maximum world depth no w1, w2, w3:World w2 in w1.next and w3 in w2.next</pre>

3.4.2 Content Constraints

Content Constraint scenarios regard restricting the contents of worlds, instead of their branch structure. These scenarios are useful to help modelers customize the simulation and facilitate the generation of particular situations. Moreover, it provides users with some upfront knowledge about the worlds that the Alloy Analyzer will generate, facilitating the cognitive task of understanding the simulation results. Furthermore, this type of scenarios provides mechanisms to check models properties, like strong/weak satisfiability, or assert the possibility of an individual exemplifying the minimum value of an association's cardinality.

In the next sections, we propose 16 content scenarios users can combine for model validation. To illustrate each scenario, we use the OntoUML diagram depicted in Figure 11, a simplification of an internship model developed by a master student in the context of a graduate course on Ontology Engineering at the Universidade Federal do Espírito Santo. The model describes that people can be students, if enrolled at educational institutions, and employees, when hired by an organization. Students may become interns at organizations, in which regular employees supervise them in their activities.

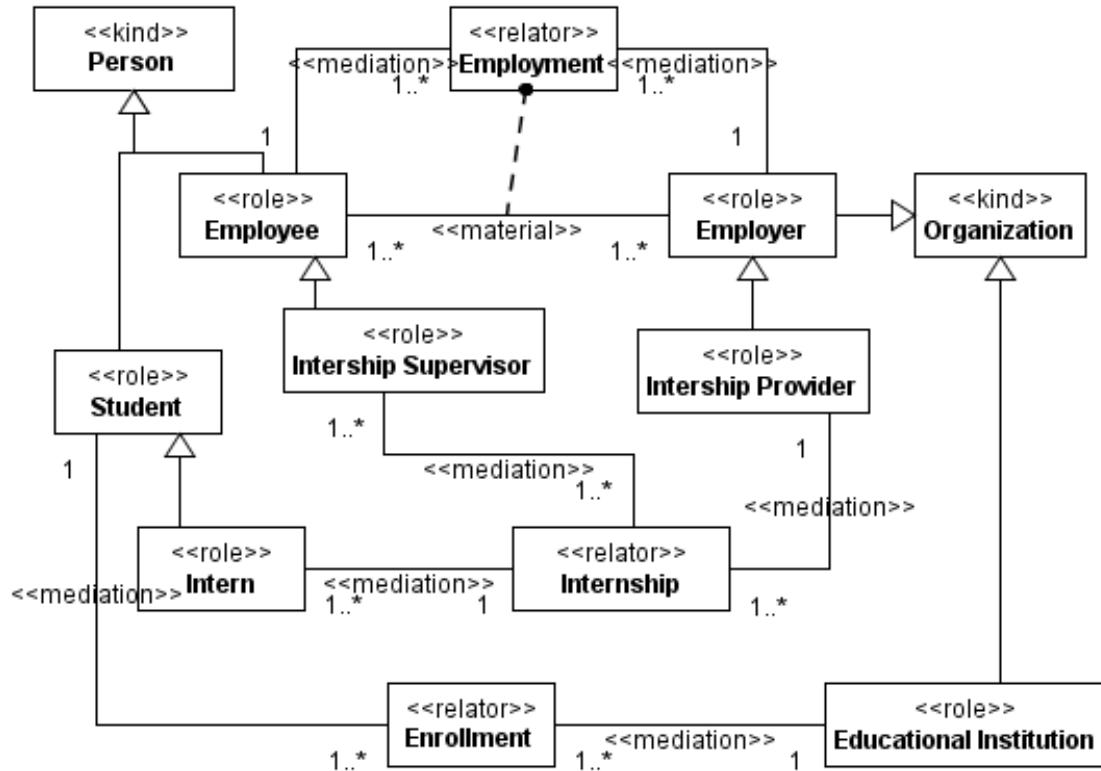


Figure 11. Internship conceptual model.

3.4.2.1 Population Size

The population of a world corresponds to the set of individuals that exist within that World, regardless of which types they instantiate (it includes both objects and properties). The population size scenario allows imposing upper and/or lower bounds for the size of a population. For instance, one may instruct the analyzer to generate worlds with at least four and at most eight individuals.

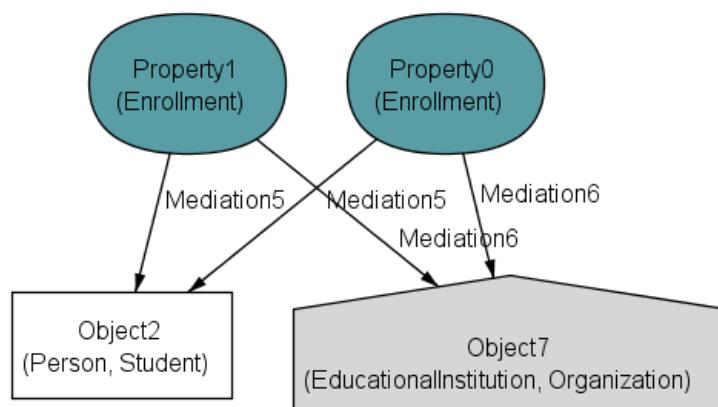


Figure 12. World with population of four individuals.

Through the usage of this scenario, one can check if the model is weakly satisfiable, i.e., is there at least one world that complies to all the ontologies constraints and has at least one instance of at least one class. Our internship model proved to be weakly satisfiable, since it has the valid finite instantiation shown in Figure 12. This world has a population of four individuals: one person (who is also a student), one organization (which is also an educational institution) and two enrollments of the same student in the same institution.

Table 13 provides the simple representation of this scenario in natural language, alongside with the Alloy expression that characterizes it.

Table 13. Population Size – scenario description.

Sentence
<i>I want to see a story with [at least / at most / exactly] [n] individuals.</i>
Alloy Expression
<code>all w: World #w.exists = n</code>

3.4.2.2 Population Variability

This scenario regards defining the variability of world population throughout the branch. One can define it as constant, where every world contains the same individuals, although they can instantiate different types. Conversely, one can set it as variable, forcing the generation of branches composed by worlds with necessarily different populations.

Although a constant population will always have the same size, it is not true that a variable population must have different sizes. Two populations are different if they do not have the same elements, and they can still do that having the same number of individuals. Thus, one can combine this scenario with the population size without generating any inconsistencies.

The main cognitive advantage of defining a constant population is that one does not need to be concerned with the dynamics of object creation and destruction. When inspecting a world, one can focus exclusively on the instantiation of anti-rigid types, or

which formal and meronymic relations are changing (material relations will not change constant because it requires the creation/destruction of new individuals: the relators).

If enforcing a whole population to be constant or variable overly restricts world generation, one can optionally do it exclusively on the object or property population. In fact, one can even combine constant object population and variable property population, which is particularly useful to analyze role playing dynamics, as well as mode and quality variations. Remember that object individuals are the ones that instantiate kinds, collectives, quantities, categories, etc., whilst property individuals are the ones that instantiate relators, modes and qualities.

To illustrate this scenario, we applied the constraint for constant population of object types to our internship example. We show, in Figure 13, the current world and in Figure 14, the future world. Note that in both worlds the objects are the same. “Object1”, “Object3” are persons (represented by the white boxes) and “Object0” and “Object2” are organizations (represented by grey pentagons).

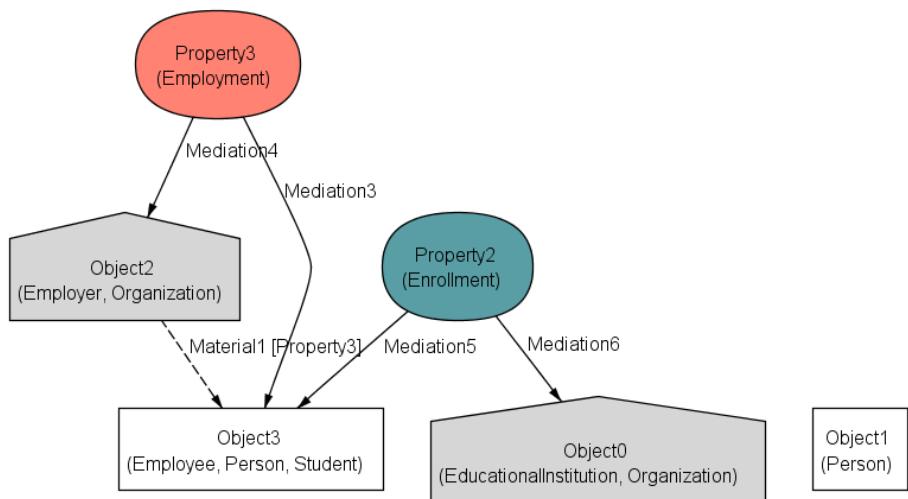


Figure 13. Population Variability: constant objects scenario - Current world.

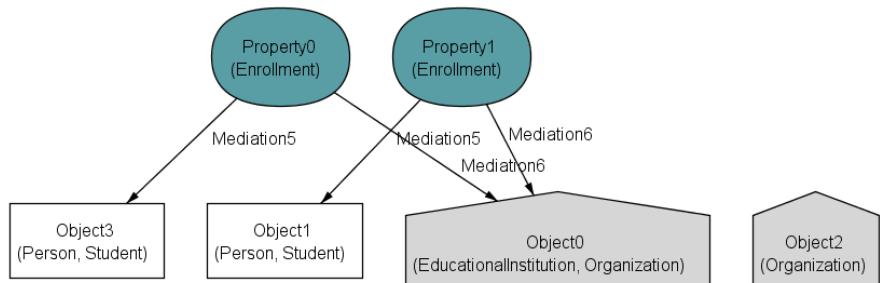


Figure 14. Population Variability: constant objects scenario - Future world.

In addition, notice that defining a constant population is equivalent to defining constant object and property populations, since all things that exist in world are either objects or properties. The same is valid for defining variable populations. What one cannot do is define complete variability at the same time as defining a partial constant population.

Table 14 provides a straightforward natural language description of this scenario with the two customizable points. The first, regarding if the population varies or not. The second, regarding which part of the world population the modeler wants to apply the constraint – the whole population, only objects or only properties. We provide the respective Alloy expression for each of the six possible combinations.

Table 14. Population Variability – scenario description.

Sentence
<i>I want to see a story where every moment [has the same / has different] [objects / properties / individuals].</i>
Alloy Expression
<pre>-- variable population all w1,w2:World w2!=w1 implies w1.exists!=w2.exists -- variable object population all w1,w2:World w2!=w1 implies w1.exists:>Object!=w2.exists:>Object -- variable property population all w1,w2:World w2!=w1 implies w1.exists:>Property!=w2.exists:>Property -- constant population all w1,w2:World w1.exists=w2.exists -- constant object population all w1,w2:World w1.exists:>Object=w2.exists:>Object -- constant property population all w1,w2:World w1.exists:>Property=w2.exists:>Property</pre>

3.4.2.3 Population Growth

The Population Growth scenario defines constraints between worlds that are directly accessible. In the incremental scenario, no individual ceases to exist in the future. For instance, if x exists in world w_0 , then in all worlds that follow it, x must also exist. That does not exclude the possibility of new things being created in future worlds, but also does not require.

Conversely, in decrement branches, we reverse this constraint: if an individual does not exist in the initial world of the branch (the one that has no predecessor) it will never come to life. From a world to its next, individuals can keep existing, but none “comes to life”. Notice that there is intersection between incremental and decremental branches: the one where every world has the same individuals. Thus, specifying a scenario as both incremental and decremental is equivalent as defining it as constant.

To exemplify, consider the two worlds depicted in Figure 15 (current world) and Figure 16 (future world), parts of the same branch. We generated these worlds by enforcing an incremental scenario. From the current to the second world, we can see the creation of two individuals: “Property0”, an instance of Employment, and “Object0”, an instance of Person and Employee. Nonetheless, all the three individuals of the current world continue to exist in the future world.

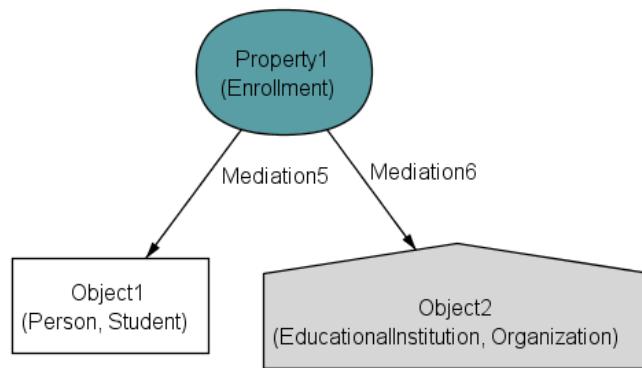


Figure 15. Population Growth (Incremental) scenario - Current World.

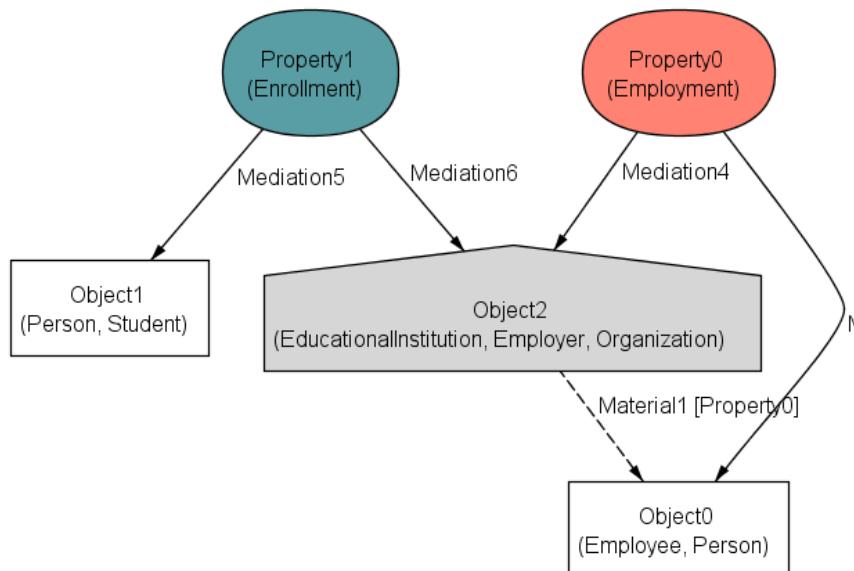


Figure 16. Population Growth (Incremental) scenario - Future World.

Table 15 provides the representation of this scenario in natural language, as if a user is demanding to see it. The only possible customization regards being incremental or decremental branch structures. Therefore, we also provide two Alloy expressions in the table, one for each case.

Table 15. Population Growth – scenario description.

Sentence
<i>I want to see a story exclusively composed by individuals [coming to existence / ceasing to exist].</i>
Alloy Expression
<pre>-- incremental_worlds all w1, w2:World w2 in w1.next implies w1.exists in w2.exists -- decremental_worlds all w1, w2:World w2 in w1.next implies w2.exists in w1.exists</pre>

3.4.2.4 Extension Size

The extension of a class in a given world corresponds to the set of individuals that are currently instantiating it. This scenario proposes the definition of lower and/or upper bounds to the size of a class' extension. Users can choose to enforce this constraint in all worlds or just a subset of them. For instance, one can require that the class Person always have exactly three objects instantiating it in every world in the generated branch.

One can check three properties with this scenario: Class Liveness, Coexistence and Strong Satisfiability. A class is “live” if, and only if, there is at least one world in which it has an instance. In order to check that, one can require the generation of at least one world in which a class' extension is greater than zero. Two or more classes coexist, if and only if, there is at least one world in which every class has at least one individual. Therefore, combining extension size constraints, one can assert coexistence. Lastly, a model is strongly satisfiable if all classes coexist, i.e., there is a world in which every class of the model has at least one instance.

To illustrate the extension size, we use the world depicted in Figure 16. In it, we see that the extension of the class “Person” is equal to two. The extensions of the classes “Enrollment”, “Employment”, “Employee”, “Student”, “Organization”, “Employer” and

“Educational Institution” are all one. Lastly, the extension of “Internship”, “Intern”, “Supervisor” and “Internship Provider” are all zero. Regarding the aforementioned properties, the figure shows that the classes “Employee” and “Student”, for example, can coexist. Furthermore, every class with a non-empty extension is also “live”.

Table 16 provides the representation of this scenario in natural language, alongside with the Alloy expression that characterizes it.

Table 16. Extension Size – scenario description.

Sentence
<i>I want to see a story composed [only / at least / at most] by worlds with [at least / at most / exactly] [n] instances of [Class].</i>
Alloy Expression
<code>all w: World #w.Class = n</code>

3.4.2.5 Temporal Extension Size

This scenario is very similar to Extension Size. The difference is that instead of defining the number of individuals that instantiate a class within a world, one can define the lower and/or upper bounds for the set containing all individuals that instantiate it throughout time. For example, if one defines a temporal extension of three for the class Person, there can only be three distinct individuals in all worlds that instantiate Person. Notice that this is not the sum of the class’ extension in each world, but the sum of distinct individuals.

To exemplify this scenario, we go back once more to the worlds depicted in Figure 15 and Figure 16. The temporal extension of “Person” is two, whilst the temporal extension of “Organization”, “Enrollment”, “Employment”, “Employer” and “Educational Institutional” are all one.

Lastly, we provide in Table 17 the representation of this scenario in natural language, alongside with the Alloy expression that characterizes it, which one can use within a fact or a predicate.

Table 17. Temporal Extension Size – scenario description.

Sentence
<i>I want to see a story with [at least / at most / exactly] [n] instances of [Class].</i>
Alloy Expression
#World.Class = n

3.4.2.6 Extension Variability

In the Extension Variability scenario, instead of setting the variability nature of the entire world population, one sets it only on the extension of a class. On one hand, enforcing a variable extension for a class implies that the individuals that instantiate it will be different for any two worlds in every generated branch. On the other hand, a constant class extension means that the same set of individuals will instantiate it in every world of the branch.

Notice that constant class extension is not rigidity, in the same way that a variable extension is not anti-rigidity. A rigid class varies its extension whenever an instance creation and destruction, because individuals cannot cease to instantiate it and still exist. An anti-rigid class, conversely, allows individuals to stop instantiating it and still exists. Therefore, it is possible to set variable and constant extensions for both kinds of classes.

Requesting a variable extension for a rigid class implies a variable population, because, as we previously explained, extensions of rigid types only change with instance creation or destruction. Thus, one cannot request variable extension for rigid classes and constant population.

Table 18 provides the representation of this scenario in natural language, alongside with the Alloy expressions that characterize it. The first expression requires “Class” to have different extensions in all worlds, whilst the second requires all extensions to be equal.

Table 18. Extension Variability – scenario description.

Sentence
<i>I want to see a story where the set of instances of [Class] [always / never] change from world to world</i>
Alloy Expression
<pre>-- variable extension all w1,w2:World w2!=w1 implies w1.Class!=w2.Class -- constant extension all w1,w2:World w1.Class=w2.Class</pre>

3.4.2.7 Extension Comparison

The extension comparison scenario includes constraints in world generating that that restrict the extensions of two types. One can require that the extension of a class is totally included in the extension of another. In addition, one can enforce disjointness, when no individual is both extensions, and equality, when every element that is one extension also is in the other.

To improve expressivity, we add the option to negate each of the three types of constraints. So we also request that a given class extension is not equal (or different), not disjoint (or overlapping) and not included in another class extension.

We illustrate this scenario with Figure 17, a world generated by the Alloy Analyzer, which defined as constraints:

- Equal extensions for classes “Student” and “Employee”
- Different extensions for classes “Student” and “Intern”
- Disjoint extensions for classes “Employer” and “Educational Institution”

Furthermore, one of the practical applications of this scenario is to verify if allegedly concrete types can have instances that are not instances of one of its subtypes. To do that, one only needs to request that the extension of a concrete class is different from the union of its subtypes’ extensions. If the simulation generates at least one world, the concrete class behaves as expect. Otherwise, it is an indication that there might be something wrong with the model.

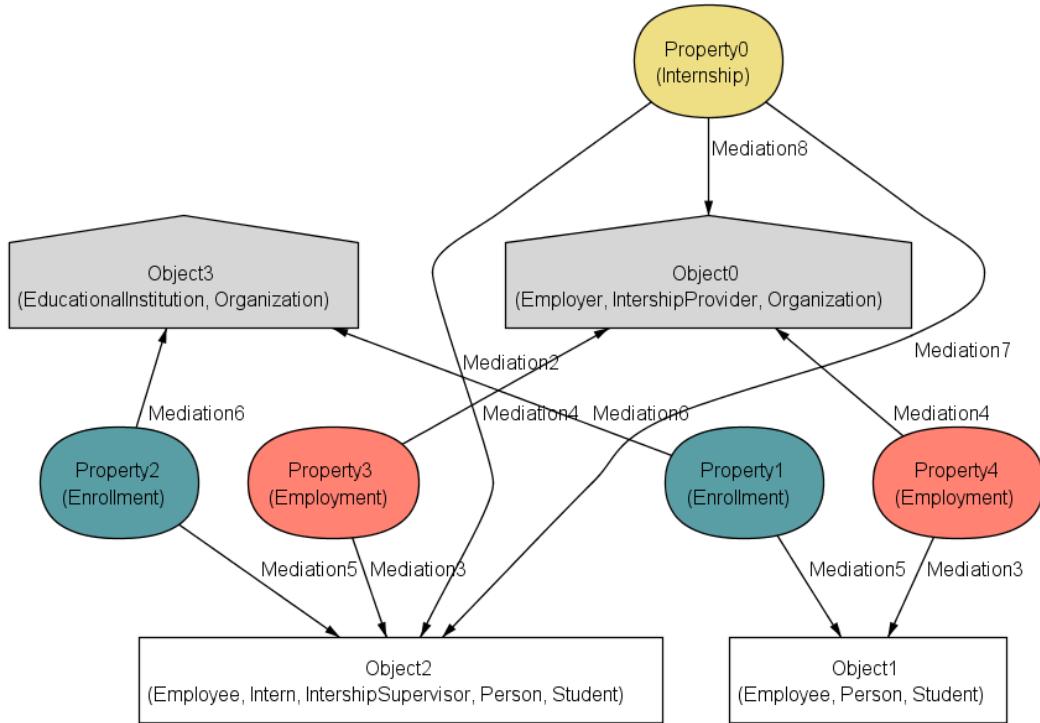


Figure 17. Extension Comparison scenario example.

Lastly, we provide, in Table 19, the representation of the Extension Comparison scenario in natural language. Moreover, we provide different Alloy expressions that characterize different assignments for the provided phrase. Particularly, we characterize subsuming, identical and disjoint extensions for classes “Class1” and “Class2”.

Table 19. Extension Comparison – scenario description.

Sentence
<i>I want to see a story where, in [all / some / at least / at most / exactly] [n] world(s), the extension of [Class1] is [not] [equal to / included in / disjoint from] the extension of [Class2]</i>
Alloy Expression
<pre>-- subsuming extensions in all worlds all w:World w.Class1 in w.Class2 -- identical extension in some world some w:World w.Class1=w.Class2 -- disjoint extension in some world #{w:World disj[w.Class1,w.Class2]}=2</pre>

3.4.2.8 Extension Size Comparison

Now, instead of comparing the individuals, we compare the size of the extensions of two classes within a world. Naturally, different sizes imply in different individuals in the extension, but same sizes do not imply in identical extensions.

To exemplify this scenario, we once more use the world depicted in Figure 17. The extension size of “Enrollment”, “Employment”, “Student” and “Employee” is the size. They are all equal to two. Conversely, the extension of “Internship” is lower than the extension of “Student”.

We provide the natural language representation of this scenario in Table 20, together with the respective Alloy expression, applicable to facts and predicates.

Table 20. Extension Size Comparison – scenario description.

Sentence
<i>I want to see a story where, in [all / some / at least / at most / exactly] [n] world(s), the number of [Class1] is [equal / greater / lesser] than the number of [Class2]</i>
Alloy Expression
-- equal extensions in all worlds <pre>all w:World #w.Class1=#w.Class2</pre>

3.4.2.9 Multiple Instantiation

The multiple instantiation scenario requires a given number of individuals to instantiate two or more types in the same world. Naturally, if one requires a multiple instantiation of two phases of the same partition, or the instantiation of two classes stereotyped as kind, the analyzer will find no results, since they are mandatorily disjoint.

This scenario can be used either to verify the possibility of a desired multiple instantiation or to verify if an undesired one has been successfully forbidden. In the former, one wants the analyzer to encounter a possible world, whilst in the second, model is correct if the analyzer does not find any.

We detail our proposal for explaining this scenario to modelers in natural language in Table 21. Note that on can customize the number of worlds ([n]) the constraint will be

applied, alongside to the number of individuals ($[m]$) target of the expression. Furthermore, we provide the Alloy expression that characterizes the scenario.

Table 21. Multiple Instantiation – scenario description.

Sentence
<i>I want to see a story where, in [at least / at most / exactly] [n] worlds, there are [at least / at most / exactly] [m] individuals that simultaneously instantiate [Class1, Class2]⁺.</i>
Alloy Expression
<code># {w : World #(w.Class1 & w.Class2)=m }=n</code>

3.4.2.10 Temporal Multiple Instantiation

The temporal multiple instantiation scenario is the temporal projection of the last scenario. It does not impose a constraint directly in a world, but in the branch as a whole. This scenario specifies that a given number of individuals must instantiate a set of classes throughout the branch. A priori, the scenario imposes no restriction regarding multiple instantiation in the same world.

Unlike the last scenario, one can require a temporal multiple instantiation of disjoint type. Phase partitions, for instance, can be exemplified when requiring at least one individual to instantiate every phase in the partition at least once. By doing that, one is able to inspect the dynamics of phase partition, i.e., in which way an individual can change phase. To exemplify, consider a phase partition of Person, composed by the phases: child, adult and elder. A person's age defines the instantiation of these phases: children are at most 17 years old, adults are from 18 to 64 and elders, 65 or older. If those are the only constraints provided in the ontology, one will still be able to born as an elder and later become a child. Table 22 provides the representation of this scenario in natural language, alongside with the Alloy expression that characterizes it.

Table 22. Temporal Multiple Instantiation – scenario description.

Sentence
<i>I want to see a story where [n] individuals instantiate [Class1, Class2]⁺ throughout time.</i>
Alloy Expression
<code># (World.Class1 & World.Class2)=n</code>

3.4.2.11 Exclusive Instantiation

This scenario intends to do the opposite of the multiple instantiation: generate worlds in which no individual simultaneously instantiate more than one of the types in the identified set. In a way, this scenario emulates a disjoint generalization set.

We obtain an interesting simulation scenario by combining exclusive instantiation and multiple temporal instantiation: an alternate instantiation branch. Naturally, this combination is only useful if the types in the set are overlapping, otherwise, the exclusive instantiation is meaningless.

Table 23 provides the representation of this scenario in natural language. Note that at the end of the sentence, we append “2+”. We intended it to show users that they must at least two anti-rigid classes must be selected. In the sequence, we provide the Alloy expression that characterizes this scenario, usable in a fact or a predicate.

Table 23. Exclusive Instantiation – scenario description.

Sentence
<i>I want to see a story where, in [all / some / at least / at most / exactly] [n] world(s), an instance of [CommonType] must instantiate only one of the following classes: [AntiRigidClass]²⁺</i>
Alloy Expression
<code>all w:World, x:w.Class (x in w.Class1 or x in w.Class2) and not (x in w.Class1 and x in w.Class2)</code>

3.4.2.12 Mandatory Anti-Rigidity

As we previously discussed, we decided not to enforce anti-rigidity by default (for more details, please refer back to section 3.5). That does not mean, however, that a user cannot enforce anti-rigidity for one or even all anti-rigid types in a particular simulation.

A user of the validation framework can apply the mandatory anti-rigidity scenario at most once for each anti-rigid class. When enforced, it means that if an individual instantiates the anti-rigid class in a given world, there is at least one other world, where the same individual exists and does not instantiate the anti-rigid class. Regardless whether the scenario has been applied for one or many anti-rigid classes, it requires a

minimum of two worlds; otherwise, the respective anti-rigid types will have no instances.

We provide our natural language description of this scenario in Table 24. The two alternative Alloy expressions that can characterize this scenario are detailed. The first, a simplified version, makes a call for the “antirigidity” predicate, defined in the ontological properties module (see Section 3.2.2). The second is a detailed version, adapted from the aforementioned predicate’s content expression.

Table 24. Mandatory Anti-Rigidity – scenario description.

Sentence
<i>I want to see a story where all objects cease to be an instance of [AntiRigidClass]</i>
Alloy Expression
<pre>-- simplified antirigidity[AntiRigidClass] -- detailed all x:class_type some disj w1,w2:World x in w1.exists and x in w1.class and x in w2.exists and x not in w2.class</pre>

3.4.2.13 Pseudo-Rigid

In the opposite direction of Mandatory Anti-Rigid, the pseudo rigid scenario emulates a rigid behavior in anti-rigid and semi-rigid classes. Meaning that if an individual instantiates an anti-rigid type in any world, it will have to do so in every other world in the generated branch, in future, past and counterfactual worlds.

This approach might seem counterintuitive at first, but it is quite useful to analyze anti-rigid classes that are subtypes of other anti-rigid classes, like a role as a subtype of a phase or a role as a subtype of another role. By emulating rigidity, one can focus exclusively in the dynamics involving the “still anti-rigid” subtypes.

Users of the validation framework should cautiously use this scenario. Otherwise, they might misinterpret the simulation results. We make only one caveat: if the user applies the fake rigid constraint for a given role or phase, the constraint will not be propagate to all its anti-rigid ancestors. What happens is that when an individual instantiates the

fake rigid class, it will also rigidly instantiate all its ancestors. Individuals that only instantiate its ancestors, though, will not be affected.

Table 25 provides the representation of this scenario in natural language. The only variability is the anti-rigid class' name. Moreover, we provide two equivalent Alloy expressions that enforce the situation described by this scenario. First, we provide a simplified version, which uses a predicate defined in our ontological properties module (see Section 3.2.2). Secondly, we provide a longer version, which uses no predicate. The expressions are equivalent.

Table 25. Pseudo-Rigid – scenario description.

Sentence
<i>I want to see a story where individuals never cease to be an instance of [AntiRigidClass]</i>
Alloy Expression
<pre>-- simplified rigidity[AntiRigidClass] -- detailed all x: World.Husband, w: World x in w.exists implies x in w.Husband</pre>

3.4.2.14 Association Changeability

The association changeability scenario deals with the dynamics of creating and destructing associations between the individuals. Although associations are undirected in OntoUML, this scenario focuses individually in a single direction of the associations for the sake of emulating dependencies between individuals. This scenario is able to emulate UFO's three types of dependencies: generic, specific and existential.

As we discussed in Chapter 2, a generic dependency between two types exist if every instance of the dependent class must be connected to at least one individual of the dependee class, but it could change throughout time. A specific dependency exists when, while an individual instantiates the dependent type, it is always connected to the same individuals of the dependee type. Lastly, the existential dependency states that while existing, the dependent individual must be connected to the same individuals of the dependee class.

To exemplify the generic dependency, at least two-world branch structures are required. The respective Alloy predicate states that a given number of individuals necessarily change the individuals they connected through the association that contains the generic dependency. We emulate specific and existential dependencies following the same logic.

To illustrate the type of worlds generated by this scenario, when configured to emulate generic dependency, we present two worlds composing the same branch. Figure 18.a depicts the current world and Figure 18.b the future one. Notice that the person identified by “Object0” works at the same organization in both world. Nonetheless, the individual has different employments with the component throughout time.

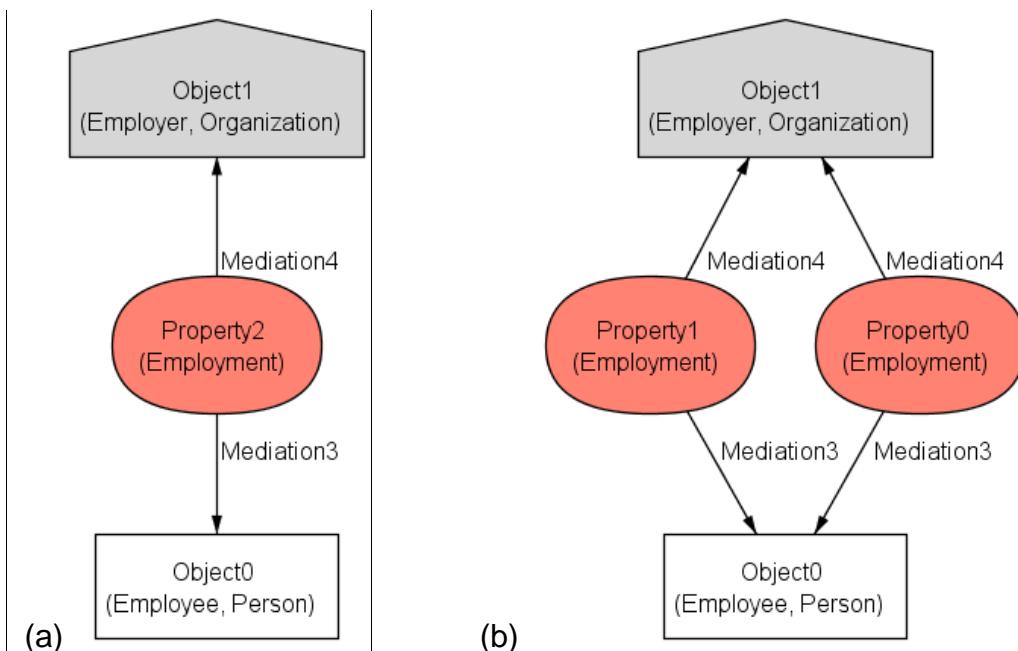


Figure 18. Association Changeability (Generic Dependency) - (a) Current World and (b) Future World

The practical utility of this scenario is three-fold. First, it has an educational utility: novice modelers can use it to understand better the difference between generic, specific and existential dependencies. Additionally, the scenario provides a way to verify whether associations in the model behave as expected. For instance, if an association should capture an existential dependency but requesting a generic dependency example produces results, there is something wrong with the ontology. Finally, it also allows users to “fix a variable” in world generation. For instance, consider that a modeler is simulating an ontology about marriage. What the modeler wants to

analyze is the variability of properties after two people get married. In this case, emulating an existential dependency between Husband and Wife is very useful.

As a final remark, we recall that the rigidity of the dependent class is relevant when simulating different types of dependency. For instance, if the dependent is rigid, a specific dependency becomes an existential dependency. In addition, if the dependent is anti-rigid, but the class defines an existential dependency, it becomes rigid.

In Table 26, we provide the representation of this scenario in natural language, identifying the variability of words using brackets. We also provide three possible Alloy expressions this scenario generates: one to emulate a generic dependency, another to simulate specific dependency and a last to simulate existential dependency.

Table 26. Association Changeability – scenario description.

Sentence
<i>I want to see a story where, in [no / every / at least / at most / exactly] world(s), [no / every / at least / at most / exactly] instance(s) of [Class1] is/are connected to the [same/different] instances of [Class2], through association [Association]</i>
Alloy Expression
<pre>-- generic dependency example, with every / every all w1,w2: World w1!=w2 implies (all x:w1.Class1 x in w2.Class1 implies x.class2[w1]!=x.class2[w2]) -- specific dependency example, with every / every all w1,w2: World all x:w1.Class1 x in w2.Class1 implies x.class2[w1]==x.class2[w2]) -- existential dependency example, with every / every all w1,w2: World all x:w1.exists x.class2[w1]==x.class2[w2])</pre>

3.4.2.15 Cardinality Value

Every association must define minimum and maximum cardinalities on both ends. However, over-constraining a model can forbid situation in which they occur. In fact, heavily over-constrained models can even prevent the instantiation of an association as a whole.

To verify the aforementioned properties on a model, we propose the Cardinality Value scenario. It consists in requesting the analyzer to generate a given number of worlds

in which $< n >$ individuals are connected to $< m >$ others through a desired association. We require users to specify this scenario individually for each association end. The $< m >$ is the tested cardinality value. For instance, considered that a simple model, composed by a class Person connected to itself through an association named “heavier than”. The cardinalities are zero or less on both ends. A possible scenario is to request that at least one world is generated in which at least one person is heavier than exactly two others are. If the Alloy Analyzer cannot find any instance, it is a suggestion that there is an over-constraining problem in the model.

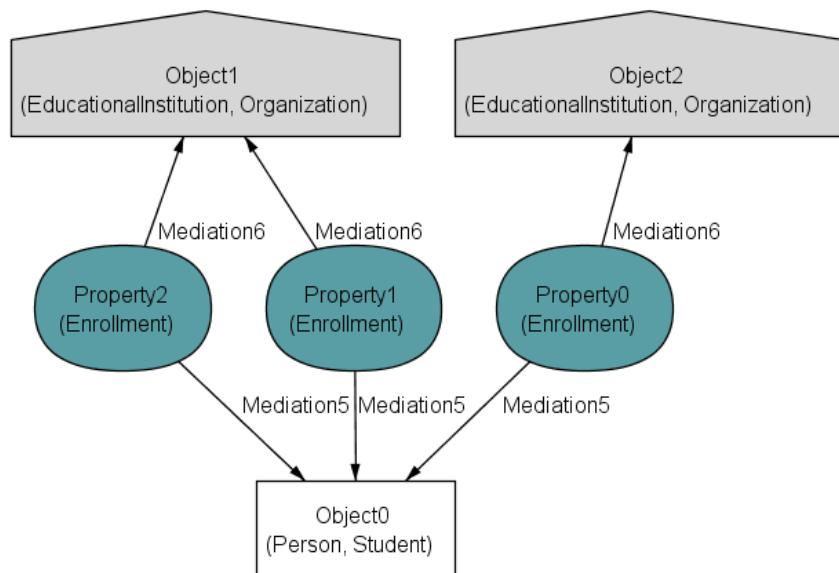


Figure 19. Cardinality Value scenario (at least three relations).

To exemplify the usage of this scenario, consider the simulation presented in Figure 19. To generate it, we defined a constraint that required every instance of “Student” to participate in exactly three enrollments. By returning this world, the analyzer showed us that the association is live, i.e., it can be instantiated finitely. Furthermore, it evinces that the model allows a cardinality of three. Table 27 provides the representation of this scenario in natural language, alongside with the respective Alloy expression.

Table 27. Cardinality Value – scenario description.

Sentence
<i>I want to see a story where, in [no / every / at least / at most / exactly] world(s), [no / every / at least / at most / exactly] instance(s) of [Class1] is/are connected to [at least / at most / exactly] [n] instances of [Class2, through association [Association]]</i>
Alloy Expression
<code>all w: World all x:w1.Class1 #x.class2[w]=3</code>

3.4.2.16 Association Depth

Analogous to the Branch Depth scenario, Association Depth prescribes a minimum and/or a maximum number of consecutive connected elements to exist in given number of worlds. It only makes sense to define this scenario for associations that connect types that can be instantiated by the same elements. In fact, the associations that can be the subject of this scenario are the same ones that characterize an anti-pattern known as BinOver (for more details, please refer to its definition in Section 5.2)

This scenario is useful to force the analyzer to generate more complex and interesting scenarios. Table 28 provides the representation of this scenario in natural language, alongside with the Alloy expression that characterizes it.

Table 28. Association Depth – scenario description.

Sentence
<i>I want to see a story where, in [no / every / at least / at most / exactly] world(s), [at most / at least / exactly] [n] instance(s) of [Class] is/are consecutively connected through [Association]</i>
Alloy Expression
<pre>-- at least one / at most / n=2 some w:World no disj x1, x2, x3: w.Class x2 in target[x1,w] and x3 in target[x2,w] -- every / at least / n=4 all w:World some disj x1, x2, x3, x4: w.Class x2 in target[x1,w] and x3 in target[x2,w] and x4 in target[x3,w]</pre>

4 ANTI-PATTERNS IN ONTOLOGY-DRIVEN CONCEPTUAL MODELING

4.1 PATTERNS, ANTI-PATTERNS AND CODE SMELLS

Design Patterns are effective standardized solutions for recurrent problems in software development. They consolidate experts' knowledge about a solution for a type of problem, making it reusable by others.

The Gang of Four's seminal work (GAMMA et al., 1994) defines Design Patterns as being a tuple composed of four elements: a pattern name, a problem, a solution and the consequences of applying the solution to that type of problem. The problem describes when to apply the pattern. The solution describes the elements that are part of the design, alongside with its properties and relationships. The solution is not static, but adaptable, such that it works in different situations. The consequences discuss the trade-offs of applying the pattern's solution to a given problem.

Inspired by the Gang of Four's work, Andrew Koenig coined the term "anti-pattern" (KOENIG, 1995). His original definition states that an anti-pattern is just like a pattern, but it produces more bad consequences than good ones. In other words, anti-patterns are wrong solutions, usually adopted, or intuitively found, but with unpleasant consequences. Patterns are usually associated to anti-patterns as better solutions to the problem at hand.

"The Blob" is an example of an anti-pattern for object-oriented software design (BROWN et al., 1998). Its authors described it as a procedural-style design, in an object-oriented environment, which cause one or few classes to aggregate most of the functionalities of the system, while the remainder classes just carry basic data or perform simple tasks.

Then there are the code smells (or bad smells). Popularized in the late 90's by Beck and Fowler in (1999), the idea of a code smell is a distinct code structure that "suggests" a further analysis because it is likely to produce maintainability and comprehensibility issues on the software being developed. The name smell conveys

the idea of symptom, that there might be something wrong. Their proposal is to use code smells as a base to refactoring the code.

The most basic example of code smell presented by the authors is the ‘Duplicate Code’. It has a quite simple definition: if you have the same code structure in two different places, your program will improve by their unification.

In the software development community, there are two views on code smells: purist and pragmatic. On one hand, the purist view perceives code smells as a certainty of a problem, using the term almost as a synonym to anti-pattern. The pragmatic view, on the other hand, perceives code smells as an indication of a possible error or bad practice, which requires particular analysis.

In this thesis, we use the notions of design patterns, anti-patterns and code smells to ontology-driven conceptual modeling (ODCM). The type of problem we deal with is formalizing into an ontology the conceptualization about a domain. The solutions are the selection and combination of the modeling language constructs for representing concepts. Inappropriate solutions, in this context, lead to domain misrepresentations.

We define anti-patterns in the context of ontology-driven conceptual modeling as the following:

Definition (semantic anti-pattern): Combination of model elements that, albeit producing syntactically valid conceptual models, is prone to be the source of domain misrepresentations. An anti-pattern must have a defined structure and refactoring options associated to it.

Note that our definition combines the ideas behind anti-patterns and code smells. On one hand, the identifiable structures capture recurrent modeling decisions that may not effectively solve the problem at hand. Moreover, combining them with appropriate solutions, our anti-pattern definition resembles the original. On the other hand, because our anti-patterns point to decisions that are not always wrong and mean to serve as a guide for modelers to validate (and refactor) their ontologies, they resemble code smells.

Do not understand our notion of anti-patterns as synonym for error or a bad practice. Think of it as model fractions that are worth “putting under the microscope”. Unlike the

traditional meaning of anti-patterns, the modeling solution is not necessarily bad. Unlike code smells, the decisions do not imply maintainability or architectural issues.

A relevant remark we make is that, for every modeling language, there should not exist a syntactically valid combination of constructs that always imply domain misrepresentation. Whenever identified such combination, it does not evidence an anti-pattern but a syntactical restriction that should be included in the language.

In the particular case of OntoUML, the incorporation of ontological constraints in its metamodel proscribes the representation of ontologically non-admissible states of affairs. However, as discussed in (BENEVIDES et al., 2010a), the language cannot guarantee that, in a particular model, only model instances representing intended state of affairs are admitted. This is because the admissibility of domain-specific states of affairs is a matter of factual knowledge, not a matter of consistent possibility (GUIZZARDI, 2010).

In the remainder of this work, whenever we use the term “anti-pattern” unqualified, we mean with the aforementioned definition, not Koenig’s original one.

4.2 TYPES OF SEMANTIC ANTI-PATTERNS

We classify the different types of anti-patterns according to the nature of the problem it might indicate, i.e., according to how the modeling decision misrepresent the domain.

A domain misrepresentation is a problem in the formalization process that generates an ontology. This process comprises materializing the conceptualization, the mental models shared by a community about a given domain, into a concrete artifact, the ontology. We are interested here in four types of formalization issues: ontological classification, scope shortcomings, under-constraining and over-constraining, which reflect the quality criteria for ontologies we discussed in Section 2.3.

One can reduce the problem of building an ontology to the problem of classifying things. From this perspective, the ontologist job is to classify concepts using meta-categories from the foundational ontology. An ontological classification issue points out a problem in this very mapping. It occurs when the meta-category used to classify a

given concept does not reflect how the experts understand that concept. In practical terms, it means the usage of the wrong construct to represent a concept.

To exemplify the aforementioned issue, consider a simple ontology about marriages. If a modeler designs the relation that exists between a husband and his wife as formal instead of material, the model contains a classification issue.

We classify anti-patterns that point to ontological classification problems as **Classification Anti-Patterns**. Note that this type of anti-pattern strongly relates to the quality criteria named Ontological Fitness.

A scope shortcoming issue evidences the need for additional concepts, relationships or properties in the ontology. A simple example would be an ontology to capture the common understanding of vehicle, which states that it is only composed of wheels, leaving out the engine, doors, bumper, etc.

If the problem “under the microscope” involves missing or unnecessary concepts or properties, we refer to it as a **Scope Anti-Pattern**. Fixing problems of this nature will obviously improve the scope quality of a model. Notice that it is not the goal of this research to evaluate the capability of the scope definition methods, but to check if there is a problem with the ontology.

The under and over constraining issues are identified through the comparison of intended and possible instantiations of the ontology. On one hand, the intended set comprises the ones that correspond to valid abstraction of states of affairs from the adopted conceptualization. On the other hand, the possible instantiations corresponds to the set of every instantiation admissible by the ontology.

An under constraining issue occurs when there is no valid state of affair that corresponds to a given instantiation. The over constraining issue is the opposite – there is no instantiation that corresponds to a valid state of affairs.

To illustrate these types of problems, consider the model on Figure 20. It describes people’s *roles* and relevant properties in the context of a criminal investigation. Some of roles may be the detectives that investigate the crime, other the suspects of committing the crime, but also witnesses that are interrogated by the detectives about the crime. Each investigation has a detective who is responsible for it. Detectives are

ranked as officers and captains. Finally, since other relational properties are relevant in investigations, the model also represents parenthood and acquaintance (“person knows person”) relations among people.

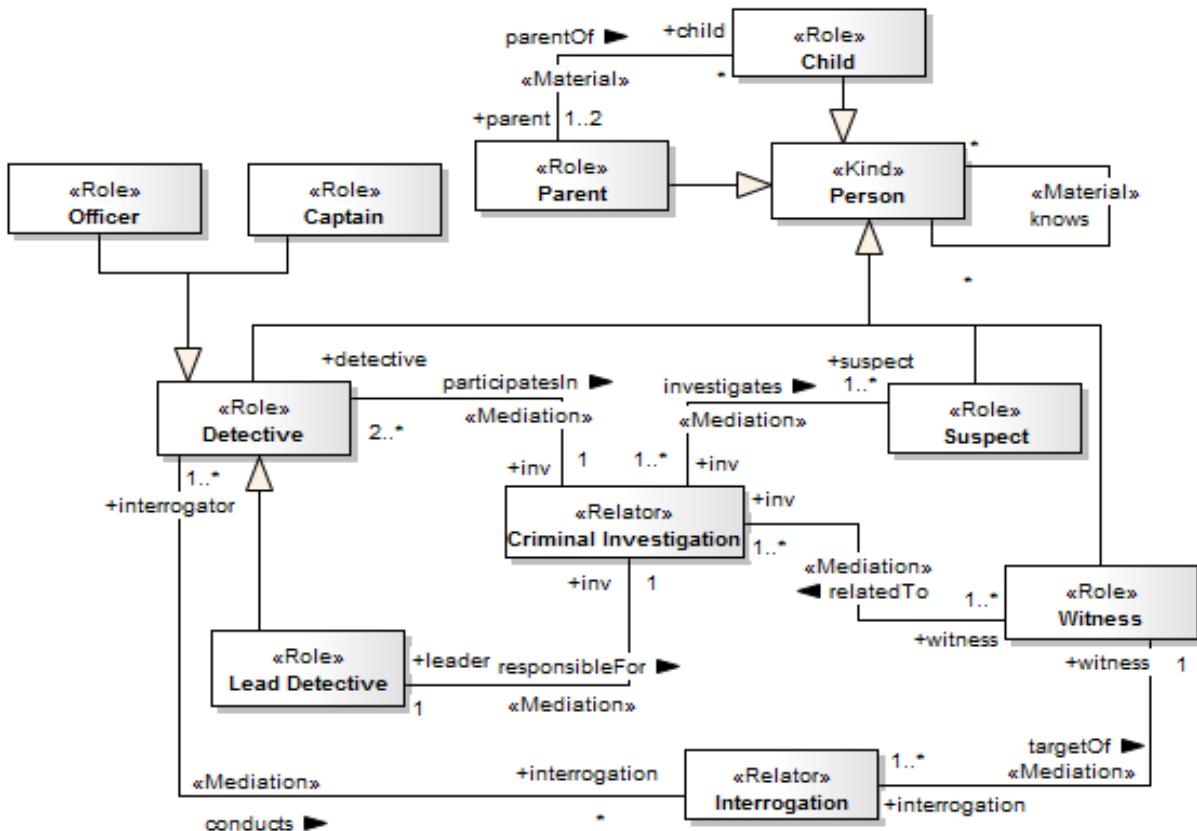


Figure 20. Partial OntoUML model of the domain of criminal investigation.

The model in Figure 20 does not violate ontological rules. It would have done so, for example, had we placed Suspect as a super-type of Person, or had we represented the possibility of a Suspect or Witness without being related to Criminal Investigation (GUIZZARDI, 2005) (we assume here a suspect is a suspect in the context of an investigation and so is a witness). However, there are still unintended model instances (according to a conceptualization assumed here for this domain) that are represented by valid instances of this model. One example is one in which the Lead Detective of an investigation is also a Suspect on that investigation. Another example is one in which a Detective interrogates himself. A third one is one in which someone is his own parent (or a parent of one of her parents).

This simple and relatively small model fragment actually contains 13 occurrences of semantic anti-patterns, but we will return to this point in Section 5. For now, what is

important is that both issues characterize what we define as **Logical Anti-Patterns**. Refactoring models based on this type of anti-pattern will improve the model's accuracy.

Note that this is not an exclusive categorization. An anti-pattern might present under constraining, scope and classification issues all together.

4.3 RELATIONSHIPS AMONG ANTI-PATTERNS

Throughout the development of the anti-pattern catalogue, we identified a set of relevant inter-pattern relationships that help characterize them individually and the repository as a whole. Besides, they are helpful in the tool support development.

Firstly, anti-patterns may share an identification logic. That means that we can design algorithms to identify occurrences of different anti-patterns using the same strategy. That is the case when the structure of the anti-patterns is similar. Identifying relations of this nature is useful for tool development, especially regarding automatizing the identification. For the users, it is useful from a learning perspective, since everything they learn from one anti-pattern applies to the other. Whenever anti-patterns share identification logics, they also share refactoring plans. Identifying this additional type of relation is useful for the same reasons.

Besides the aforementioned grouping, we relate anti-pattern w.r.t. the type of element or structural feature they analyze. For example, some anti-patterns are relator-centered, whilst others are hierarchy-centered. Knowing this characteristic for an anti-pattern is useful for two purposes. First, it suggests an order of investigation, e.g. hierarchy problems should precede to relator, or meronymic-centered issues. Second, because it allows one to use anti-patterns in combination with other methods to validate a particular aspect of the model. For instance, a modeler could use meronymic-centered anti-patterns in combination with meronymic analysis patterns (GUIZZARDI, 2009) to validate the mereological aspects of her ontology.

Furthermore, we unveiled that adopting a particular solution for an anti-pattern occurrence might have the side effect of fixing or extinguishing another occurrence –

of the same anti-pattern type or not. For instance, if by fixing an occurrence one deletes a relator, all other anti-patterns occurrences in which the deleted relator participated would disappear. This beneficial side effect indicates that, by adopting an order, anti-pattern analysis might become more efficient.

Lastly, we identified a causality relationship amongst anti-patterns, i.e., the solution of one might generate an occurrence of another. In some cases, the generated anti-pattern does not require further analysis, since it will always be a false positive. Conversely, by fixing an anti-pattern, one might generate another as side effect. In those cases, one can understand the analysis of the second anti-pattern as an extension of the first one.

4.4 THE ROLE OF ANTI-PATTERNS IN ONTOLOGY VALIDATION

As we discussed in the motivation section of this thesis, our main goal is to develop an ontology validation framework for managers. That means that we want to provide modelers with validation tools that require as minimum learning overhead as possible. Anti-patterns play a central role in this framework.

Just like design patterns for software development, anti-patterns are a solution to reuse knowledge in conceptual modeling. In this case, instead of reusing certified solutions, we reuse recurrent problems and appropriate solutions for these problems.

We propose to use anti-patterns in the following manner: first, the structures defined for each anti-pattern provide modelers a starting point in the validation process, i.e., after the modeling activity the anti-patterns confront them with their potentially problematic decisions. Secondly, the analysis flow defined for each anti-pattern guides the user through the process of review his decision, reaching a conclusion if it was appropriate one or not. Finally, the refactoring plans provide systematic instructions on how to modify their model in order to reach the desired outcome. Our proposal takes inspiration in (HARTMANN, 2001) in the sense of defining three steps for improving the model: problem detection, analysis and refactoring.

We do not claim our anti-pattern catalogue to be an exhaustive set of all recurrent “dangerous” conceptual modeling decisions. We argue that verifying and refactoring an ontology using the catalogue as a guide will improve the ontology’s precision, coverage, scope and classification. It is not, and should not be the only validation technique applied.

Logical Anti-Patterns have the additional role of unveiling instance-level domain constraints, i.e., aid the modeler in the exploration of the domain by making her think of unusual situations allowed by the model. This will be clearer throughout the explanation of the catalogue in the next chapter, but suffice to say that the anti-patterns will unveil constraints that OntoUML constructs cannot express. Without the anti-patterns, modelers are most likely not even to ask some questions to the domain experts.

Finally, as we will discuss in detail in Chapter 8, with proper tool support, we argue that modelers can use anti-patterns without costly learning requirements. We fully automate the identification step, thus requiring no learning from the user side. Furthermore, we implement a wizard tool to guide the analysis flow. Lastly, the tool automatically runs pre-defined refactoring plans.

5 THE ANTI-PATTERN CATALOGUE

In this chapter, we present our catalogue of semantic anti-patterns for ontology-driven conceptual modeling.

In order to facilitate learning, usage and comparison of the anti-patterns, we describe them following a consistent format. We discuss each anti-pattern initially in a textual unstructured way. In the sequence, for each anti-pattern, we present a table that summarizes the most important information discussed.

The template we use to describe anti-patterns follows:

- **Name:** uniquely identifies the anti-pattern and intends to convey a brief idea of its content.
- **Acronym:** a short name to facilitate the documentation and communication about the anti-pattern.
- **Description:** a natural language description of the generic structure that characterizes the anti-pattern. It also presents required constraints to characterize the anti-pattern occurrence, when necessary.
- **Justification:** a brief discussion of why modelers should “put under the microscope” the model structure identified by the anti-pattern.
- **Type:** identifies the types of the anti-pattern, which indicates the type of problem the structure suggests. The possible values are:
 - *Logical*, for under and over constraining issues,
 - *Scope*, for missing or unnecessary constructs; and
 - *Classification*, for errors in the choice of the meta-category used to represent a concept;
- **Feature:** indicates the element of the OntoUML’s meta-model that is in the center of the anti-pattern. For instance, some anti-patterns focus on Relators, others on Mixins, and so on.
- **Structure:** formal description that characterizes an occurrence of the anti-pattern. Consists of pattern roles, constraints and a diagrammatic generic

example. Note that some anti-patterns have one or more structures, which we call variations².

- *Pattern Roles*: describes the elements that participate in the anti-pattern, their possible stereotypes and cardinalities. We give each pattern role a proper name.
- *Constraints*: Logical expressions that must always be true to characterize an occurrence of the anti-pattern. Constraints can be general, involving multiple roles, or role-specific.
- *Generic Example*: a figure that exemplifies the generic structure of the anti-pattern. When the anti-pattern has structural variations, we present multiple figures.
- **Refactoring Plan:** Every anti-pattern must define a set of refactoring plans. These plans define a sequence of actions that modify the model in order to fix the domain misrepresentation issue. Some plans may be mutually exclusive, if they cannot be performed in the context of a single occurrence, or complementary, if they can. Note that some plans are only applicable to certain variations of the anti-pattern (identified by the tag [conditional]). The refactoring plans are composed mainly by the following types of actions:
 - *Create Constraint (OCL)*: indicate the definition of addition of OCL invariants or derivation rules (e.g. making explicit how a relation is derived or forbidding instances to relate in a certain conditions).
 - *Modify Element (Mod)*: indicate a change in a model element. The most frequent ones are stereotype changes (e.g. from Formal to Material or from Collective to Kind) and meta-property changes (e.g. *isReadOnly* from false to true).
 - *New Element (New)*: indicate the creation of model elements.
 - *Delete Element (Del)*: indicate the destruction of model elements.
- **Anti-Pattern Relations:** indicate different types of relations between anti-patterns (for a more elaborate discussion, please refer back to Section 4.3). The types of relations are:

² Some patterns contain multiple structures because they all generate the same type of problem and we can fix them using very similar actions.

- *Group by Feature*: indicate the anti-patterns that analyze the same feature of the current anti-pattern.
- *Group by Type*: indicates the anti-patterns that have the same type of the current anti-pattern.
- *Causes*: indicates that performing one of the proposed refactoring options might generate an occurrence of another anti-pattern.
- *Caused by*: indicates the anti-patterns whose refactoring plans can generate the current anti-pattern. It is the inverse of the previous one.

For each anti-pattern, we developed an **analysis flow**. We designed these flows as UML Activity Diagrams and they contain questions and pre-defined possible answers to guide the modeler in deciding whether an occurrence of the anti-pattern indeed is a mistake. At the end of the questionnaire, in addition to providing the previous answer, the flow identifies the appropriate refactoring solution and describes the steps to perform it. These analysis flows are the “blueprints” for the anti-pattern wizard we implemented (see Section 8.5). Due to organization reasons, we present them in Appendix B.

Occurrences of an anti-pattern correspond to a particular set of elements represented in a particular model that “fit” the structure of an anti-pattern.

To exemplify the structure of the anti-patterns, we present **examples** encountered in real models. In order to show how to analyze and refactor models using the anti-patterns, we analyze the presented examples and discuss particular solutions for each case. Note that these examples are not included in the anti-pattern summary table.

5.1 ASSOCIATION CYCLE (AssCyc)

The Association Cycle (AssCyc) occurs when an arbitrary number of types are connected through the same number of relations in a way that composes a cycle (in the same meaning defined in Graph Theory). In other words, one can start navigating relations from any type in the cycle and arrive back to the starting point without going through the same relation and visiting the same type more than once (except the

first/last). Notice that we intentionally use the term “relation”, because we mean both associations (material, formal, componentOf and so on) and generalizations.

We argue in favor of analyzing this model structure because it allows two very characteristic instantiations scenarios: one in which there are cycles at the instance level, and another one where there is not. Our studies showed that usually, only one type of scenario could occur.

Not all cycles, though, are occurrences of this anti-pattern. The first requirement is that two or more associations must compose it. Cycles exclusively composed of generalizations are excluded, because they are either a syntactical issue (when all generalizations are in the same direction) that should be forbidden, or just regular hierarchies. Cycles with only one association are most likely characterization of another anti-pattern, named *BinOver*, while with two association they usually characterize occurrences of *RelSpec* or *RelOver*.

The second constraint that must hold, is that an occurrence cannot be exactly a characterization of the Relator Design Pattern (GUIZZARDI, 2005), i.e., one of the types is a relator that connects two other types by mediations, who are connected between themselves by a material. In addition, it cannot for a very simple reason: the derivation from the relator to the material relation imposes closed cycles at the instance level. For the same reason Derivations are not allowed because they are always used in the same way, connecting a relator to a material, and thus, always form cycles.

The last constraint that must hold for a proper characterization of the AssCyc anti-pattern is that every association in the cycle must be intentional. This requirement is justified because the semantic variability (open or closed instance-level cycles) has already been addressed by the derivation rule(s) created by the modeler.

We propose three refactoring plans: first, to enforce the open cycle instantiation scenario at instance level through the specification of an OCL invariant; second, is an analogous solution to forbid instance level cycles; third, one of the associations is set as derived and its derivation OCL rule specified.

Table 29 summarizes the description of the AssCyc anti-pattern.

Table 29. Characterization of the AssCyc anti-pattern.

Name (Acronym)		Description
Association Cycle (AssCyc)		This anti-pattern occurs when n types are connected through n relations forming a cycle, i.e. one can start navigating relations from a type and arrive back without using the same relation more than once and without visiting the same type more than once (except the first/last).
Type	Feature	Justification
Logical	Association	The analysis of two characteristic instantiation scenarios: one in which there are cycles at the instance level, and another one where there is not.
Pattern Roles		
Mult.	Name	Possible Types
3..*	Rel-n	Association (all but «derivation ») or Generalization
3..*	Type-n	Class
Constraints		
<ol style="list-style-type: none"> 1. The relations must form a cycle, i.e., starting from every Type_n, one can navigate the relations and arrive back to the same type using every relation exactly one time. 2. The number of associations in the cycle must be greater than 2. 3. Every association must be intentional (isDerived=false) 4. The cycle cannot characterize a Relator Design Pattern. 		
Generic Example		
<pre> classDiagram class Type4 class Type1 class Type2 class Type3 Type4 "3..*" "3..*" Type1 : +type3, +type1 Type1 "3..*" "3..*" Type2 : -type1, -type2 Type2 "3..*" Type3 : +type2 Type3 "3..*" "3..*" Type4 : -type4 </pre>		
Refactoring Plans		
<ol style="list-style-type: none"> 1. [OCL] Enforce cycles: create OCL invariant to <u>enforce</u> instance level cycles according to following template (any type can be used as base): <pre> context Type-1 inv: self.type2oclAsType(Type-3).type4asset()->includes(self) </pre> 2. [OCL] Forbid cycles: create OCL invariant to <u>forbid</u> instance level cycles according to the following template (any type can be used as base): <pre> context Type-1 inv: self.type2oclAsType(Type-3).type4asset()->excludes(self) </pre> 		

3. **[Mod/OCL] Derive association:** set the selected association as derived and create an OCL derivation rule. Suggest template below:

```
context Type-1 :: type3 : Bag(Type-3)
derive: self.type2.oc1AsType(Type-3).type4.asSet() = self.asSet()
```

Anti-Pattern Relations

Group by Feature (Association): BinOver, ImpAbs, RelComp, RelSpec

Group by Type (Logical): BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Causes: none

Caused by: MultDep

To exemplify the AssCyc anti-pattern, consider the following fragment of the O3 ontology (PEREIRA; ALMEIDA, 2014) depicted in Figure 21. The ontology describes a subset of the organizational domain. The most distinguished concepts are:

- *Formal Organization*, like a company or a university;
 - *Organizational Unit*, which can be understood as departments of an organization;
 - *Employee Type*, which captures the notion of what is commonly referred to as position, the official work post, like professor or manager;
 - *Business Role* is a class that formalizes the idea of particular functions or roles, played by members of an organization. Examples are PhD supervisor and tutor.

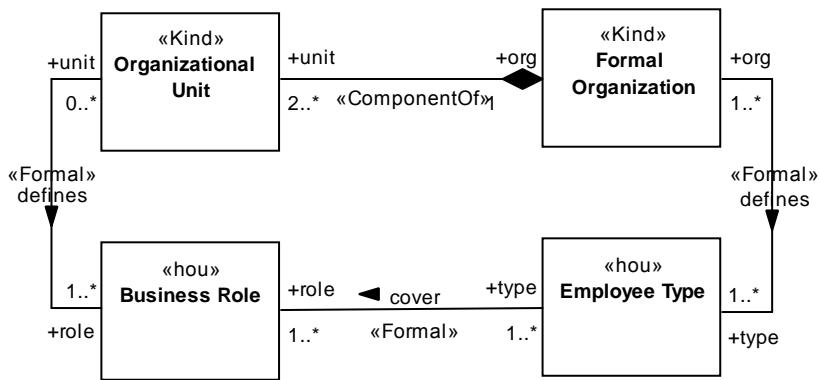


Figure 21. Fragment of the O3 ontology that contains an occurrence of the AssCyc anti-pattern.

The relations state that an organization has two or more units; a unit defines roles that employee can play; an organization defines positions to which people can be hired into; and that positions implied the possibility of playing certain business roles.

As aforementioned, the structure identified in Figure 21 allows the instantiations of open and closed cycles at the instance-level. Figure 22 presents a model instance that characterizes an open cycle. Notice that the organization defines an employee type (position) that covers business roles defined by organization units that are part of another organization.

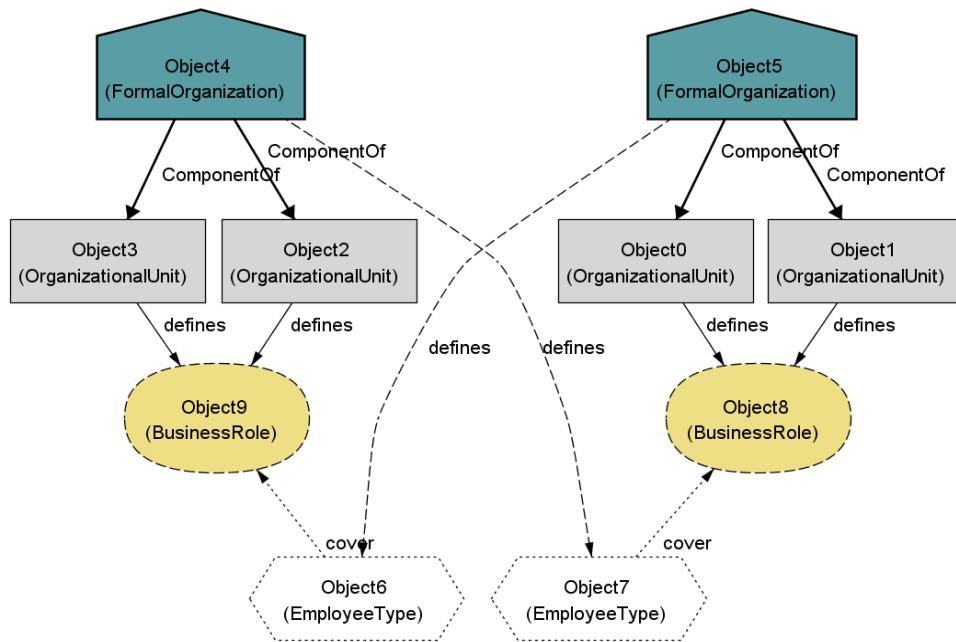


Figure 22. A possible instantiation of the O3 ontology, exemplifying an open instance cycle.

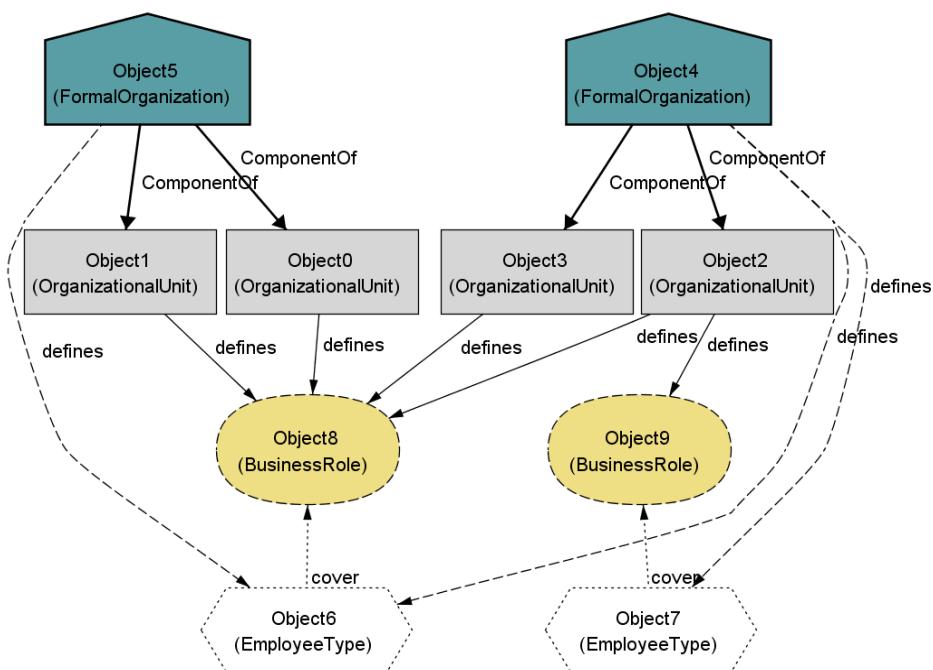


Figure 23. A possible instantiation of the O3 ontology, exemplifying a closed instance cycle.

In another way, Figure 23 presents an instantiation that characterizes a closed cycle. Note that in this case, if a unit defines a business role, the organization it composes must define an employee type that covers such position.

When confronted with both situations, the authors of O3 agreed that only instances like the one in are valid. Since they also concluded that all associations in the identified cycle are intentional, we propose the OCL invariant in Listing 4 to make the model more precise:

Listing 4. OCL invariant generated to enforce closed cycles in the O3 fragment.

```
context 'Formal Organization'  
inv acyclic: self.unit.role.type.org->includes (self)
```

5.2 BINARY RELATION BETWEEN OVERLAPPING TYPES (BINOVER)

To describe this anti-pattern we first define the concepts of *overlapping* and *disjoint* set of types.

On one hand, a set of types is said to be **overlapping** if, and only if, there is at least one possible instantiation of the model in which at least one individual simultaneously instantiate all types in the set. The set containing Employer, Male and Adult types, for example, are overlapping, since there are many adult male workers in the world. We formally state this definition as follows:

Definition (Overlapping Set): Let W be a non-empty set of possible worlds, $w \in W$ be a specific world, T the set of types, $t \in T$ be a particular type, $ext_w(t)$ ³ the extension of a t in world w and $exists(w)$ the function that return all individuals that exists in a world w . A set of types is overlapping if there is at least one w , such that:

$$\forall t, t' \in T, \exists x, x \in exists(w) \wedge x \in ext_w(t) \wedge x \in ext_w(t')$$

Conversely, we classify a set of types as **disjoint** if, and only if, there is no possible instantiation in which an individual instantiate more than one type in the set. As an

³ We use the function $ext_w(t)$ as defined in (GUIZZARDI, 2005)

example, consider the types Adult, Child and Elder. No individual, at any point of time, can instantiate all these types simultaneously.

Definition (Disjoint Set): Making the same conventions as in the previous definition, a set of types is disjoint, if for every w :

$$\forall t, t' \in T, t \neq t' \rightarrow \nexists x, x \in \text{exists}(w) \wedge x \in \text{ext}_w(t) \wedge x \in \text{ext}_w(t')$$

The Binary Relation Between Overlapping Types (BinOver) corresponds to an association, of any stereotype, that connects two types that compose an overlapping set. It means that the same individual may instantiate both ends of the relationship. A given relation $\langle R \rangle$ between types $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ characterize a *BinOver* occurrence when:

1. $\langle \text{Source} \rangle$ equals $\langle \text{Target} \rangle$
2. $\langle \text{Source} \rangle$ is a direct or indirect subtype of $\langle \text{Target} \rangle$;
3. $\langle \text{Target} \rangle$ is a direct or indirect subtype of $\langle \text{Source} \rangle$;
4. $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ are sortals (Subkind, Role or Phase) that share a common identity provider (Kind, Collective, Quantity) and there is no generalization set which makes them explicitly;
5. $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ are relators that share a common super-type and there is no generalization set which makes them explicitly disjoint;
6. $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ are modes that share a common super-type and there is no generalization set which makes them explicitly disjoint;
7. $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ are mixins (Category, Mixin or RoleMixin) that directly or indirectly generalize at least one common sortal (Kind, Quantity, Collective, Subkind, Role, Phase);
8. $\langle \text{Source} \rangle$ and $\langle \text{Target} \rangle$ are mixins (Category, Mixin or RoleMixin) that share a common mixin super-type and none of their subtypes are sortals;

In our preliminary study, we reported structures (1) and (4) as being two different anti-patterns, named Self-Type Relationship and Binary Relation Between Overlapping Subtypes respectively (SALES; BARCELOS; GUIZZARDI, 2012). After conducting further analysis, we decided to merge and expand them into the anti-pattern presented in this section. The reasoning for it is that, although they are different in structure, both

anti-patterns convey the same conceptual problems and the same set of solutions is admissible.

The focus on the aforementioned structures arises from the fact that it is not always intuitive that the related types overlap. When they do, we learned that is useful to specify binary relation properties, like reflexivity or transitivity, in order to prevent misrepresentations of the domain.

From our empirical studies, we learned that the most useful binary properties for conceptual modeling are reflexivity, symmetry, transitivity and cyclicity. We formally define each property in Table 30. For a detailed listing and description of more complex binary properties, please refer to (SCHMIDT; STROHLEIN, 1993).

Table 30. Relevant binary properties for conceptual modeling.

Binary Property	Definition	Example
Reflexive	$\forall x \in X \rightarrow R(x, x)$	is equal to
Antireflexive	$\forall x \in X \rightarrow \neg R(x, x)$	is father of
Symmetric	$\forall x, y \in X, R(x, y) \rightarrow R(y, x)$	is brother of, is married to
Antisymmetric	$\forall x, y \in X, R(x, y) \wedge R(y, x) \rightarrow x = y$	greater or equal to (\geq)
Transitive	$\forall x, y, z \in X, R(x, y) \wedge R(y, z) \rightarrow R(x, z)$	is ancestor of
Acyclic	$\forall x_1, x_2, \dots, x_n \in X, R(x_1, x_2) \wedge R(x_2, x_3) \wedge \dots \wedge R(x_{n-1}, x_n) \rightarrow \neg R(x_n, x_1)$	is ancestor of

The association's stereotype heavily influences the possible refactoring alternatives the modeler can apply to association. For some types of relations, such as characterizations, mediations and part-whole relations, the language already embeds binary properties, whilst on others, such as material and formal relations, these constraints are defined by the modeler (or not). Table 31 presents the embedded binary properties values for the stereotypes of associations in OntoUML. In addition to the basic binary properties, we identify in the last column of the table if the language allows modelers to define associations with that particular stereotype connecting a class to itself.

Table 31. Binary property values embedded in OntoUML's associations.

Stereotype	Reflexivity	Symmetry	Transitivity	Cyclicity	Type-Reflexive
Formal	Undefined	undefined	undefined	undefined	Allowed
Material	Undefined	undefined	undefined	undefined	Allowed
Mediation	n.a.	n.a.	n.a.	n.a.	Forbidden
Characterization	Irreflexive	Asymmetric	n.a.	Acyclic	Forbidden
ComponentOf	Irreflexive	Asymmetric	Transitive	Acyclic	Allowed
MemberOf	Irreflexive	Asymmetric	Intransitive	Acyclic	Forbidden
SubCollectionOf	Irreflexive	Asymmetric	Transitive	Acyclic	Allowed
SubQuantityOf	Irreflexive	Asymmetric	Transitive	Acyclic	Forbidden

We propose three refactoring alternatives for a BinOver occurrence: change the association's stereotype, create OCL invariants to enforce a desired binary property and “force” the related types to be disjoint. Note that, if a modeler enforces the related types to be disjoint, she will not be able to set any binary property, since the relation will no longer have the same individuals in the domain and range.

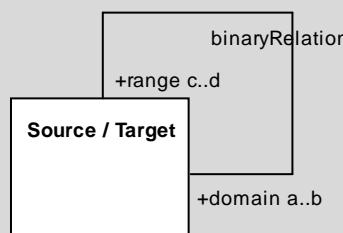
Table 32 summarizes the description of the *BinOver* anti-pattern

Table 32. Characterization of the *BinOver* anti-pattern.

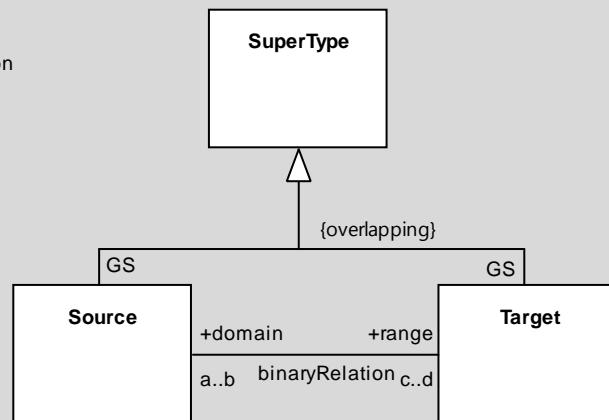
Name (Acronym)		Description
Binary Relation between Overlapping Types (BinOver)		A binary relation whose end types are overlapping characterizes this anti-pattern.
Type	Feature	Justification
Logical	Association	Modelers often do not perceive by themselves that two or more types overlap. This anti-pattern makes them aware of that and confronts modelers with the possibility to specify binary relation properties, like reflexivity, transitivity and symmetry.
Pattern Roles		
Mult.	Name	Possible Types
1	binaryRelation	Association (all but «derivation »)
1	Source	Class
1	Target	Class

Generic Example*

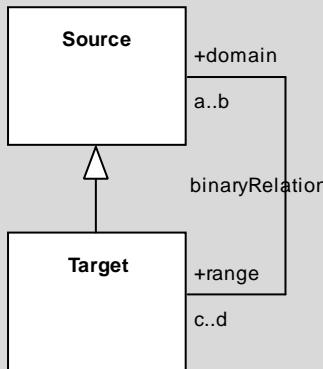
Variation 1: Source equals Target



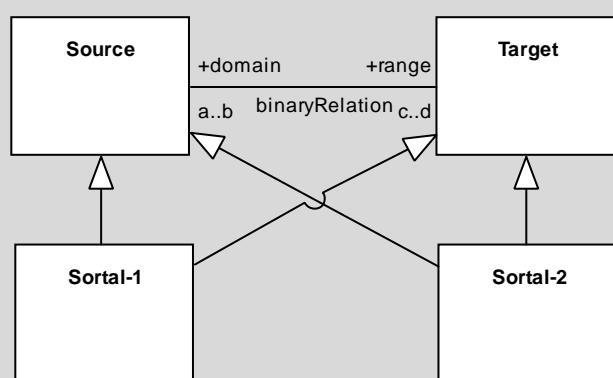
Variation 4: Overlapping Subtypes



Variation 2: Target subsets Source



Variation 5: Overlapping Mixins (Common Sortals)



*Note: the presented variations are illustrative and do not intend to cover all possibilities

Refactoring Plans

1. **[Mod] Fix stereotype:** change the stereotype of the relation to fit a desired binary property
2. **[OCL] Enforce binary property:** create OCL invariant to enforce a desired binary property (as long as it is compatible with the embedded constraints of the stereotype)
3. **[New] Enforce disjointness:** make the related types disjoint by the specification of a disjoint generalization set.

Anti-Pattern Relations

Group by Feature (Association): AssCyc, ImpAbs, RelComp, RelSpec

Group by Type (Logical): AssCyc, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Caused by: none

Causes: none

We extracted and adapted the small model fragment depicted in Figure 24 from the MGIC Ontology, in order to exemplify the BinOver anti-pattern. It regards Railway Systems, i.e., collections of railways that the Brazilian government concedes to private

companies. These systems are connected to one another, which the model formalizes as the “isConnectedTo” formal association.

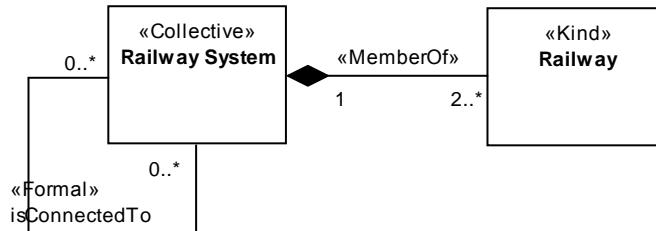


Figure 24. Fragment of the MGIC that exemplifies BinOver.

This example fits the first structural possibility assigned for BinOver, an association that connects a type to itself. Figure 25 depicts two possible worlds. On the left, built using white boxes, is a representation of a possible world in which the relation of being connected is transitive and acyclic. On the right, composed by grey boxes, is a possible world in which the relation is both symmetric, reflexive, transitive and cyclic. In this case, the modelers chose to create a rule for make the relation symmetric and reflexive.

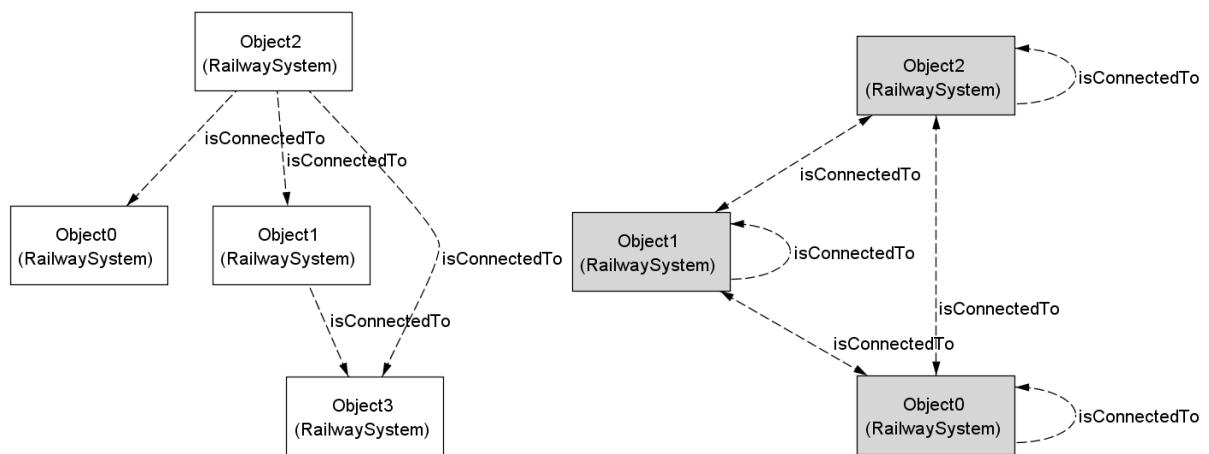


Figure 25. On the left, in white, a possible world in which the “isConnectedTo” relation is transitive and acyclic; on the right, in grey, a world where “isConnectedTo” is reflexive, symmetric and cyclic

5.3 DECEIVING INTERSECTION (DECINT)

An occurrence of the Deceiving Intersection anti-pattern occurs when a class, stereotyped as Subkind, Role, Phase, Mode or Relator, specializes two or more concrete types.

We use the concept of *concrete type* to refer to types that can have instances that are not instances of any of its child types. From a structural perspective, that implies two things: first, the meta-attribute *isAbstract* of the concrete type must be set to false, and second, the meta-attribute *isCovering* of every generalization set which aggregates generalizations leading to the concrete type, must be set to false. Note that the definition of concrete type automatically rules out all three mixin classes, since the language requires that their *isAbstract* attribute is always set to false;

The main driver to investigate this particular model structure is to help the modeler decide whether the subtype with multiple generalizations is intentional or derived by the intersection.

On one hand, *intentional subtyping* refers to the characterization of subtypes by the addition of complementary characteristics. On the other hand, *derived subtyping* refers to the characterization of subtypes by evaluating a set of properties of the parent type (e.g. the restriction of a quality value). To exemplify, consider the following types: Physical Object, which classifies every individual that have volume and mass; Colored Object, which classifies physical objects that are opaque and thus have a Color quality; and Black Object, used to qualify black things. By stating the Colored Object is a subtype of Physical Object, we are intentional subtyping, i.e. adding the characteristic of having the color quality. By stating that Black Object is a subtype of Colored Object, we are deriving: selecting every object that has a black color.

Derivation by intersection (OLIVÉ, 2007) is a particular type of derivation subtyping, which states that if an individual instantiates every type in a pre-determined set, it also instantiates the subtype. For example, consider the type Employee, which represents people that have formal jobs, and the types Adult and Underage, which describes people over and under 18 years old respectively. If one desires to represent the concept of Underage Employee, i.e., the type of 18 years old or younger employees,

derivation by intersection is the solution. To address this constraint, we propose the creation of an OCL invariant (template provided in Table 33)

In complement to the main motivation, the auxiliary reason to investigate this anti-pattern is to verify if the multiple inheritance does not generate an empty extension for the type. That occurs when two or more parent types disjoint due to a generalization set. It also occurs when two or more parent types provide/inherit different identity principles (this condition only applies for sortals classes, naturally).

Table 33 summarizes the description of the Declnt anti-pattern.

Table 33. Characterization of the Declnt anti-pattern.

Name (Acronym)		Description
Deceiving Intersection (Declnt)		An occurrence of the Declnt anti-pattern occurs when a type specializes two or more concrete types
Type	Feature	Justification
Logical	Hierarchy	Investigate if the subtype with multiple generalizations is intentional or derived by the intersection (main) and if its extension is not empty.
Pattern Roles		
Mult.	Name	Possible Types
1	Type	«subkind», «phase», «role», «mode» or «relator»
2..*	Parent-n	All class stereotypes but «mixin», «roleMixin» and «category»
Constraints		
<ol style="list-style-type: none"> 1. The specialization of the parents into Type must be syntactically valid, e.g. if type is a relator, all its parents must also be relators. 2. There must be at least two parents for which the following conditions evaluate to true: <ol style="list-style-type: none"> a. Parent_n.isAbstract = false b. For all gs : Generalization Set whose common supertype is Parent_n, gs.isCovering=true 		
Generic Example		
<pre> graph TD Type --> Parent1[Parent-1] Type --> Parent2[Parent-2] Type --> Parent3[Parent-3] </pre>		

Refactoring Plans

1. **[conditional] [Mod] Fix Generalization Set:** can only be adopted if two or more parent types are made disjoint by a generalization set. The possible solutions are to remove the existing generalization set or set its *isCovering* property to true.
2. **[conditional] [Mod] Fix Identity Principle:** can only be applied if Type is sortal (Subkind, role or phase) and they do not follow the same identity principle. The action consists on defining the single identity provider.
3. **[Mod/Del] Invert/Delete Generalization:** consists of deleting and/or inverting one or more generalizations from Type to one of the identified parents.
4. **[OCL] Derived by Intersection:** create an OCL derivation or invariant constraint to specify that the extension of type is derived by the intersection of the extensions of two or more concrete parents:

```
context Parent1
inv: (self.oclIsTypeOf(Parent2) and self.oclIsTypeOf(Parent2))
implies self.oclIsTypeOf(Type)
```

Anti-Pattern Relations

Group by Feature (Hierarchy): GenSet, MixIden, MixRig, UndefPhase

Group by Type (Logical): AssCyc, BinOver, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Causes: none

Caused by: none

Adapted from a model that formalizes FIFA's official football rules, Figure 26 depicts a Declnt occurrence. The fragment partially describes the expulsion event (the famous red card) and states that a referee can expel both players and team officials (e.g. coach, doctor). It tries to simulate the RoleMixin pattern for disjoint roles that follow the same identity.

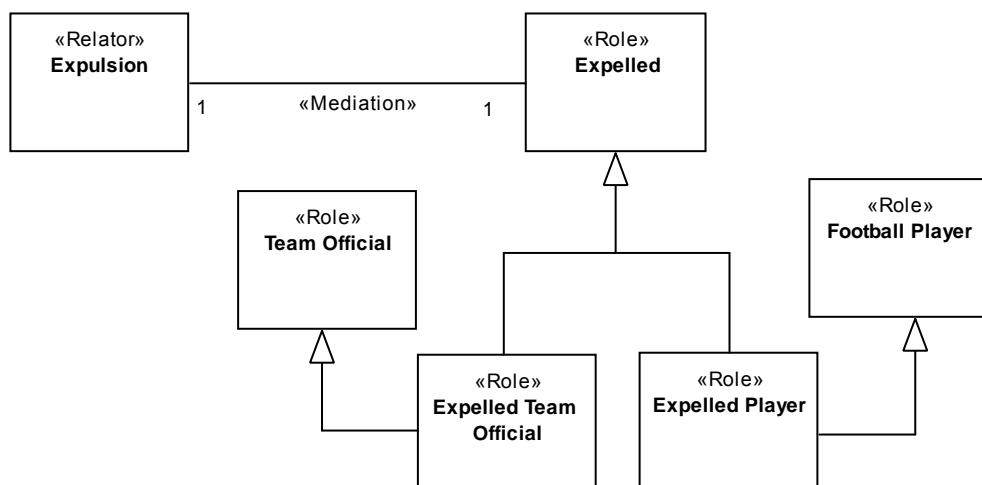


Figure 26. Simplified fragment of the FIFA Football Model characterizing a Declnt occurrence.

By naming the roles as “Expelled Team Official” and “Expelled Player”, the modeler already “gives away” the intention of subtyping through intersection. Nonetheless, the model still allows a person to play the roles defined by “Expelled Member” and “Football Player” and not be an “Expelled Player”.

One can solve this particular problem in more than one way. A simple solution to this particular case would be to set Expelled as abstract. Another would be to create a complete generalization set containing the generalizations from Expelled Team Official and Expelled Player to Expelled. Nonetheless, these solutions cannot be generically applied to every occurrence of the Declnt anti-pattern and thus, by default, the suggestion is to create the OCL invariant presented in Listing 5.

Listing 5. OCL invariant enforcing derivation by intersection.

```
context FootballPlayer
inv: self.oclisTypeOf(Expelled) implies self.oclisTypeOf(ExpelledPlayer)
```

5.4 RELATIONALLY DEPENDENT PHASE (DEPPhase)

As described in Chapter 2, phases and roles capture anti-rigid types, whose instances share the same identity principle. The main difference between them is that, on one hand, phases are instantiated when there is a change in an intrinsic property, such as a quality (e.g. age, color, weight) or a mode (e.g. disease, intention). On the other hand, roles are instantiated when there is an establishment of a relational property (e.g. becoming married, a student or a parent). OntoUML formalizes relational properties, in this sense, through associations stereotyped as mediation.

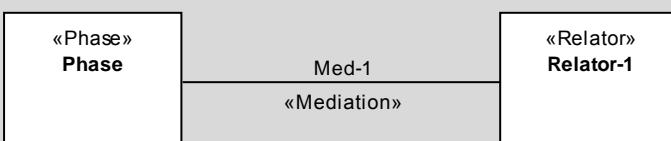
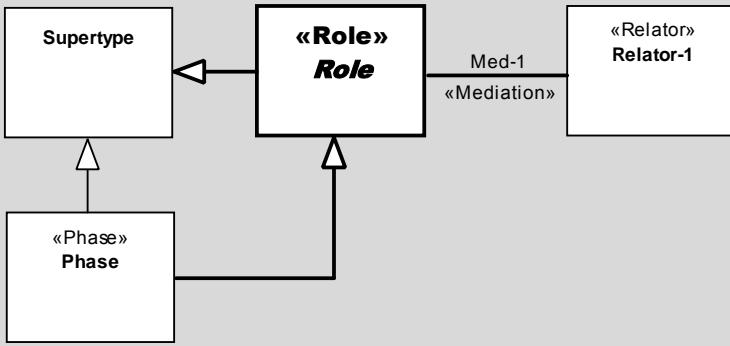
Mixing the concept of phase and role is what generates an occurrence of the Relationally Dependent Phase (*DepPhase*) anti-pattern. Its simple identification structure, a phase connected to a mediation, is a hint that the model may be wrong.

Three possible conclusions can arise through the analysis of a *DepPhase* occurrence. First, the modeler can conclude that she qualified the phase with the wrong stereotype and, in fact, it should be a role. Second, modelers can conclude that the phase does not own, but inherits the relational dependency. Reaching this conclusion leads to the creation of a new role type to act as the parent type of the phase. The last possible

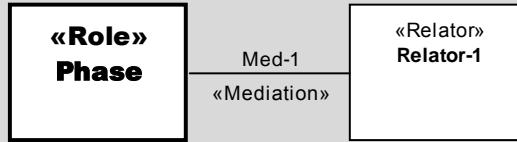
conclusion is that the phase is characterized both by a change in an intrinsic property and the establishment of a relational property. In this last case, a modeler should keep the model as it is and the occurrence is a “false alarm”.

Table 34 consolidates the description of the DepPhase anti-pattern.

Table 34. Characterization of the DepPhase anti-pattern.

Name (Acronym)		Description
Relationally Dependent Phase (DepPhase)		A class stereotyped as «phase» connected to one or more «mediation» associations.
Type	Feature	Justification
Classification; Scope	Phase; Relator	Phases are instantiated when there is a change in an intrinsic property. Roles are instantiated when there is a change in a relational property. Selecting the phase stereotype for a class but connecting it to a mediation is “mixing up” the two meta-categories.
Pattern Roles		
Mult.	Name	Possible Types
1	Phase	«phase»
1..*	Med-n	«mediation»
1..*	Relator-n	«relator»
Generic Example		
		
Refactoring Plans		
<ol style="list-style-type: none"> 1. [New/Mod] Make the role explicit: Create a «role» as a parent type of the phase and move the mediation it. 		
		

2. [Mod] Change phase to role: Change the stereotype of the phase to «role»



Anti-Pattern Relations

Group by Feature (Phase): UndefPhase

Group by Feature (Relator): FreeRole, MultDep, RelOver, RelRig, RepRel

Group by Type (Classification): GSRig, HetColl, HomoFunc, MixIden, MixRig, RelRig, UndefFormal, UndefPhase

Group by Type (Scope): FreeRole, ImpAbs, MultDep, GSRig, HomoFunc, MixIden, MixRig, RelRig, UndefPhase

Causes: none

Caused by: none

We use a structure found in the Quality Assurance Model to exemplify an occurrence of the DepPhase anti-pattern. The model, as a whole, describes concepts and properties relevant to process evaluation within organizations. There are standard processes defined by the organization, executed in particular projects. The appraiser evaluates these executions according to a set of criteria and identifies lessons learned, non-compliant items and problems.

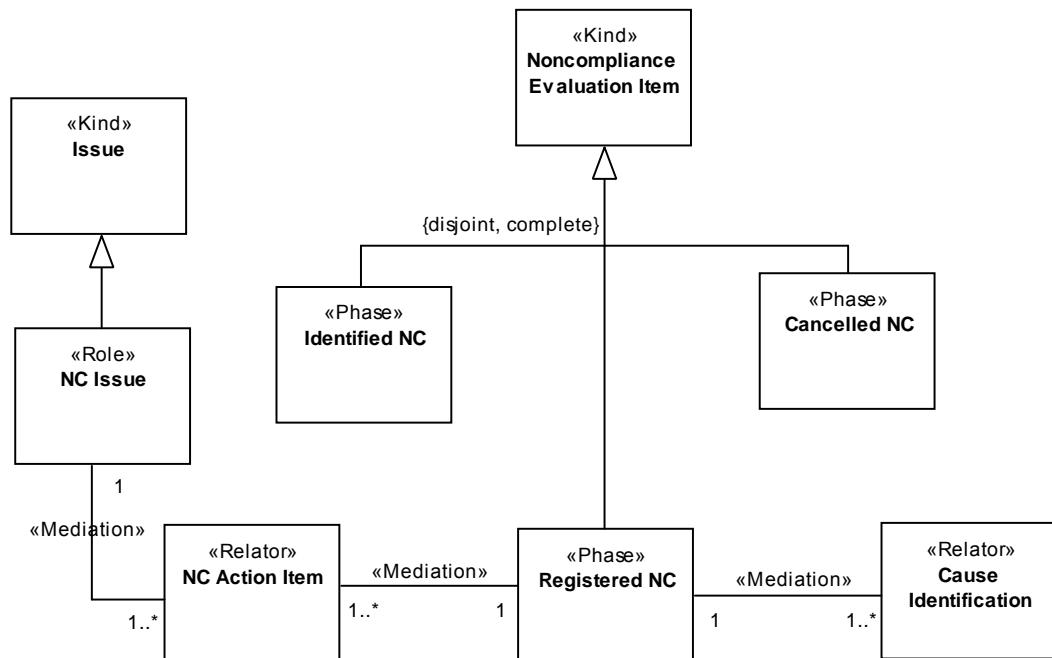


Figure 27. DepPhase occurrence identified in the Quality Assurance Model.

The simplified fragment in Figure 27 describes the categorization of the non-compliant items in the process. Note that there are 3 phases for such items: Identified, Cancelled and Registered, the latter being the relationally dependent phase. A non-compliant item becomes registered because of the identification of the cause and the creation of an issue, as represented by the model. The conclusion reached, thus, is that the registered phase is in fact a role.

As a last remark, note the other two phases, identified and cancelled, also should not be classified as such. Cancelled represented items that were the target of a cancellation action. Identified is a type derived by exclusion (OLIVÉ, 2007), i.e. items that are not registered nor cancelled. The representation of the latter is in fact optional. As an alternative, one could model the generalization set as incomplete.

5.5 FREE ROLE SPECIALIZATION (FREEROLE)

The Free Role Specialization (FreeRole) anti-pattern occurs when a «Role» type connected to a «Relator» through a «Mediation» association, is specialized in other «Role» types, which do not directly own an additional «Mediation» association.

As discussed in Chapter 2, roles are externally dependent types. That means that every role type must be directly or indirectly connected to a mediation, which defines it. The difference between direct and indirect connection is that, in the former, the type itself is connected, whilst in the latter one of its ancestors types is directly connected. The focus of this anti-pattern is on these indirect connected roles, or as we call them, free roles.

The goal of the analysis is to identify instantiation condition for all free-roles. To help in this process, we propose four alternative role specialization patterns, namely: derived sub-role, intentional sub-role, material sub-role and role of role.

The **derived sub-role** pattern applies to free-roles when they are instantiated according to a pre-determined set of conditions. For example, a person plays the role of student when she enrolls at an educational institution. Students are freshman if their

enrollment is at most one year old. If this example, the “Student” class would be defined as a role, whilst the “Freshman” class, a derived sub-role.

The second alternative solution is setting the free role as a **role of role**, which assumes the specification of a new and independent relator to characterize the role specialization. A modeler should take this path when the event that gives rise to the relator that characterizes the free role is different from the one that characterizes the defined role. To exemplify, consider again the student concept. A person, when playing the student role, can become an intern, but to become one, they need a company to accept them in an internship, the independent relator.

Notice that, a priori, an instance of the defined role type can become and cease to be an instance of the free roles defined as a derived sub-role or a role of role, while being connected to the same instance of the defining relator.

The next role specialization pattern, named **intentional sub-role**, consists in specializing the defining relator and connecting it to the respective sub-role. To exemplify it, we once more go back to the student concept. Now, consider the concepts of undergraduate and graduate students. Even though both imply being a student, and thus, having an enrollment, they each require enrollments with particular characteristics. Graduate enrollments, for example, require (or at least assume the possibility of) a supervisor assignment.

Lastly, we propose the **material sub-role** pattern, which modelers should apply to the free role when a particular subset of the defining relator defines it, but this particular subset does not imply in the addition of extra characteristics. To exemplify in the educational domain, consider that, in order to obtain their titles, graduate students must make a presentation of their thesis to a group of professors, the thesis committee. After the presentations, the committee provides a verdict: approved or failed. Notice that in both cases, the model does not provide additional properties for the relator “Verdict”, only if it defines an approved thesis or a failed one. The difference from the material sub-role pattern to OntoUML’s traditional representation of roles is that, in the former the material relation is derived from a relator connected to a direct or indirect parent type of the role, whilst on the latter, the material is derived from a relator directly connected to the role.

Differently from the first two patterns, a modeler should apply the last two when she wants to enforce an individual to always instantiate the free role while being connected to a particular instance of the defining relator.

For didactical reasons, instead of showing an example encountered in one of the models of our repository. Figure 28 presents the application of all proposed role specialization patterns, using the examples previously discussed in the educational domain. To improve readability, we made some adaptations in the diagram. We hid the kind “Person” (super type of the roles “Student” and “Professor”), the relator between “Graduate Student” and “Thesis” and the roles of “Company” and “Educational Institution”. Furthermore, we represent the roles that characterize the application of the proposed patterns with thicker lines. A dashed line indicates the pattern type they represent.

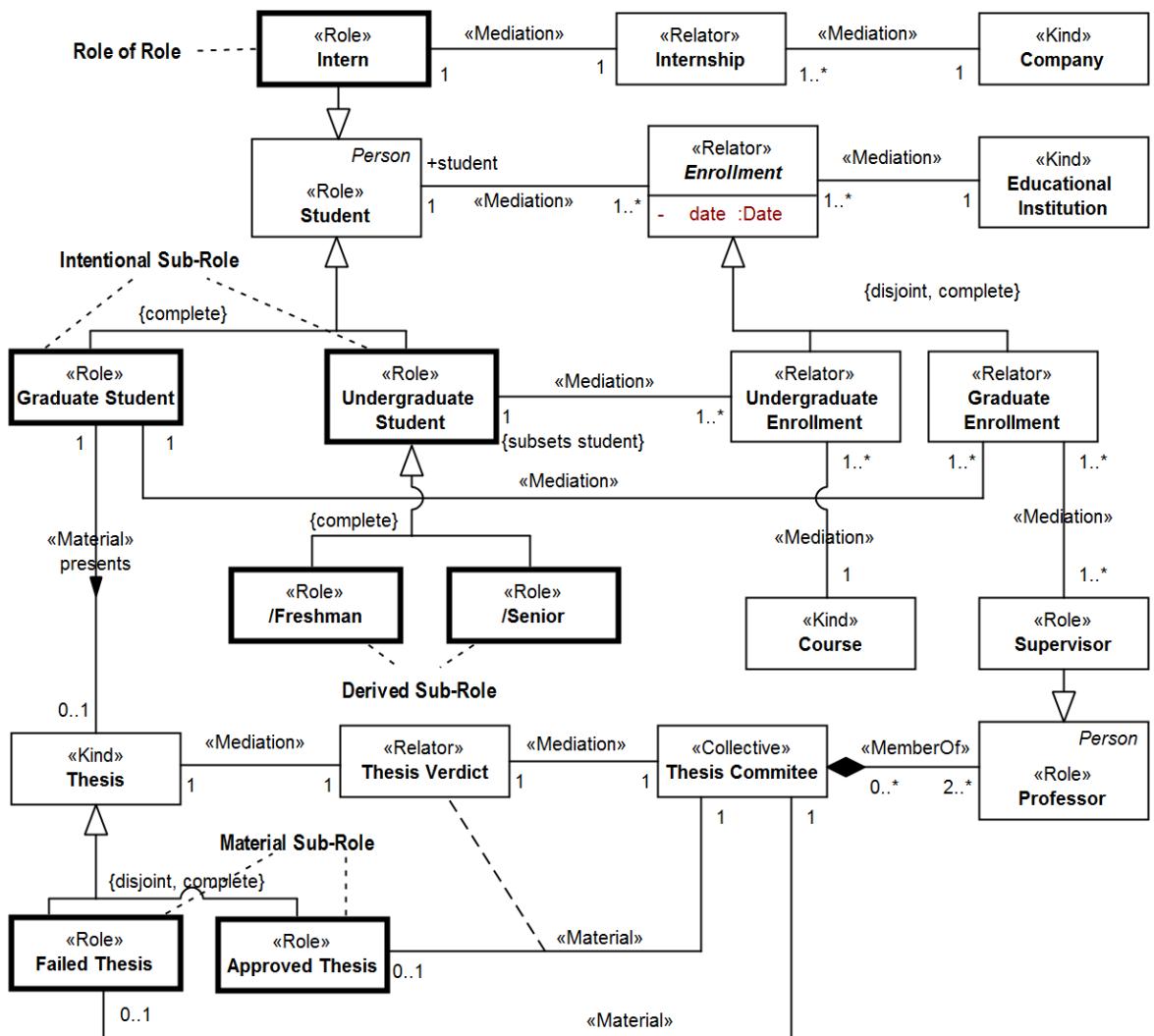


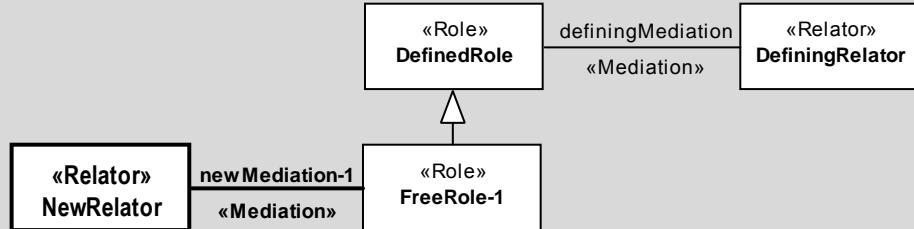
Figure 28. Application of the four role specialization patterns.

Table 35 consolidates the description of the FreeRole anti-pattern.

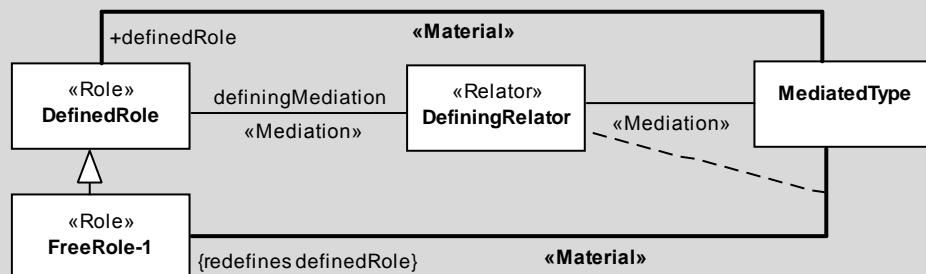
Table 35. Characterization of the *FreeRole* anti-pattern.

Name (Acronym)		Description
Free Role Specialization (FreeRole)		A «role» type connected to a «relator» type through a «mediation» association, is specialized in one or more «role» types, which in turn are not connected to an additional «mediation» association
Type	Feature	Justification
Logical; Scope	Role; Relator	Identify the condition required for the instantiation of the subtypes of the role that are not connected to any relator, since no particular condition was defined.
Pattern Roles		
Mult.	Name	Possible Types
1	DefinedRole	«role»
1..*	definingMediation-n	«mediation»
1..*	DefiningRelator-n	«relator»
1..*	FreeRole-n	«role»
Constraints		
Every free role must meet the following requirements:		
<ol style="list-style-type: none"> 1. It cannot be directly connected to any mediation 2. It cannot be a direct or indirect subtype of a RoleMixin that is directly connected to a mediation from a hierarchy path that does not go through DefinedRole. 		
Generic Example		
<pre> classDiagram class DefinedRole { <<Role>> } class FreeRole_1 { <<Role>> } class DefiningRelator { <<Relator>> } DefinedRole "1" -- "1..*" DefiningRelator : definingMediation DefinedRole "*" -- "1..*" FreeRole_1 : <<Mediation>> FreeRole_1 "*" -- "1..*" DefiningRelator : <<Relator>> </pre>		
Refactoring Plans		
<ol style="list-style-type: none"> 1. [OCL] Set derived role as derived: The instantiation of a free role defined by a derivation rule, which can be defined as follows: <pre> context FreeRole-1 :: allInstances() : Set(FreeRole-1) derive : DefinedRole.allInstances() ->select(x <CONDITION>) </pre> 		

2. **[New] Add independent relator:** a free role is defined by another relator which has no relation to DefiningRelator. Implies the creation of a relator and a mediation, like in the structure:



3. **[New] Add a redefining material relation:** a free role is defined by a redefining material relation, like in the structure:



Anti-Pattern Relations

Group by Feature (Role): none

Group by Feature (Relator): RelOver, RelRig, RepRel, UndefPhase

Group by Type (Logical): GSRig, HetColl, HomoFunc, Mixlden, MixRig, RelRig, UndefFormal, UndefPhase

Group by Type (Scope): DepPhase, ImpAbs, MultDep, GSRig, HomoFunc, Mixlden, MixRig, RelRig, UndefPhase

Caused by: none

Causes: none

5.6 GENERALIZATION SET WITH MIXED RIGIDITY (GSRIG)

Generalization Sets (GS) impose disjointness and completeness constraints in a group of generalizations that lead to the same parent type. Disjoint sets are the ones that forbid an individual to instantiate more than one subtype, whilst complete sets require an instance of the parent type to instantiate at least one of the subtypes in the set.

A GS should only aggregate generalizations that follow a common specialization criterion, which is the type of property used to define why an instance of a type

becomes an instance of one of its subtypes. To clarify, consider the types Person, Man, Woman, Child and Adult. People are classified as Man or Woman according to their gender, whilst they are classified as Child or Adult through the evaluation of their age.

The Generalization Set With Mixed Rigidity (GSRig) anti-pattern aims to identify structures that suggest the usage of two or more specialization criteria in a single GS, leading to classification or scope issues. The GSRig's identification structure is a GS whose common parent type is rigid (stereotyped as «kind», «quantity», «collective», «subkind» and «category») and that has at least one generalization coming from a rigid type and one from an anti-rigid type (stereotyped as «phase», «role» and «roleMixin»).

Although proposed to address classification and scope issues, a particular GSRig structure leads to inconsistency issues. This special case occurs when a disjoint and complete GS contains one or more rigid subtypes and exactly one anti-rigid subtype. The disjointness constraints, imposes that no individual created as an instance of one of the rigid subtypes will ever instantiate the anti-rigid type. The *isCovering* meta-property set to true imposes that no individual, which instantiates the anti-rigid type since its creation, will ever cease to do so. In other words, the anti-rigid subtype “becomes” rigid.

To determine if a *GSRig* occurrence indeed characterizes a modeling error, the modeler should start by analyzing the rigidity of the subtypes. If after analyzing, and possibly changing the subtypes' stereotypes, all of them turn out to be rigid or anti-rigid, the anti-pattern is fixed.

The next investigation path focus on the common parent's stereotype, but a modeler should only consider it if the subtypes inherit/provide different identity principles. Having rigid and anti-rigid subtypes is one of things that define a Mixin and it might be the case that the modeler should have used it to qualify the parent type.

The third refactoring proposal starts by investigating the specialization criterion used for each subtype. If the modeler concludes that she used more than one, the occurrence characterizes a mistake. Therefore, the solution is to create additional GSs and move the generalizations accordingly.

The last refactoring plan is to create rigid subtypes that are the new direct parents of one or more anti-rigid subtypes. If the modeler opts to create only one rigid subtype, she can optionally set it as derived by the negation of the other rigid subtypes.

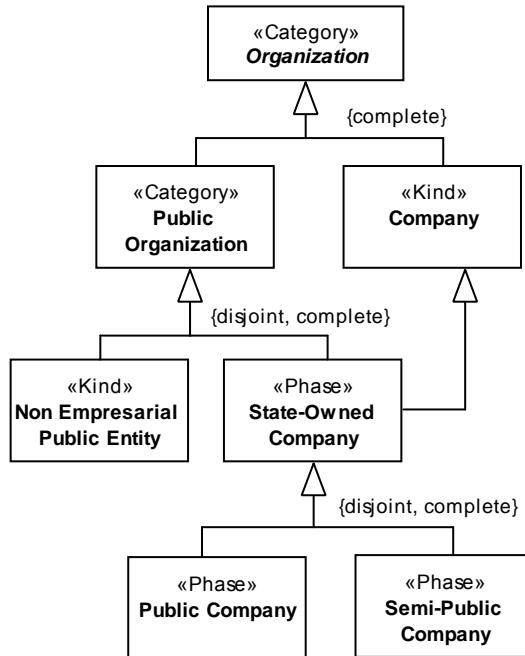


Figure 29. Translated and simplified fragment of the MPOG Ontology Draft exemplifying the GSRig anti-pattern

Now, we provide a *GSRig* example obtained from an ontology draft made by the Ministério do Planejamento, Orçamento e Gestão⁴ (MPOG), which discusses governmental view on organizational structures. Figure 29 depicts a hierarchy formalized in the draft that starts with the organization concept, the most generic classification they provide. In its refinements the concepts of public organizations (owned by the government) and companies (owned by the private sector) arise, the former being further refined in non-empresarial public organizations, like a ministry, and state-owned companies, like Infraero (responsible for managing Brazilian airports) and Petrobras (an oil and gas company). The former is an example of a fully public company (only the Brazilian government manages it) and the latter, of a semi-public one (a shared enterprise between the private and the public sectors).

The GS that enforces every instance of “Public Organization” to be either a “Non Empresarial Public Entity” or a “State-Owned Company” characterizes the GSRig

⁴ In English: Ministry of Planning, Budget and Management

occurrence. In fact, this occurrence also characterizes the logical inconsistency we previously mentioned, forcing phase to be rigid. The solution, though, is quite simple: change the stereotype of Public Organization and Organization to «mixin».

Table 36 summarizes the main contents of the *GSRig* anti-pattern.

Table 36. Characterization of the *GSRig* anti-pattern.

Name (Acronym)		Description
Generalization Set with Mixed Rigidity (<i>GSRig</i>)		A generalization set whose common super-type is rigid and from all its generalizations, at least one comes from an anti-rigid type and at least one comes from a rigid type.
Type	Feature	Justification
Classification; Scope	Hierarchy; Gen. Set	Generalization sets groups generalizations leading to a common super-type, all defined using the same specialization criterion. If the super type is not a mixin and the subtypes have different rigidity properties, they probably do not belong in the same generalization set.
Pattern Roles		
Mult.	Name	Possible Types
1	GenSet	Generalization Set
1	RigidParent	«kind», «quantity», «collective», «subkind» and «category»
1..*	Rigid-n	«kind», «quantity», «collective», «subkind» and «category»
1..*	AntiRigid-n	«phase», «role» and «roleMixin»
Generic Example		
<small>*Note: stereotypes are only illustrative</small>		
Refactoring Plans		
<ol style="list-style-type: none"> [Mod] Fix subtype rigidity: choose the option if you conclude that one or more stereotype of the subtypes is wrong. Change them to achieve only rigid or anti-rigid subtypes for the generalization set. [New/Mod] Split generalization set: the generalization set aggregates multiple specialization criteria. Create additional generalization sets and move the respective generalizations. 		

3. **[New/Mod] Implicit rigid subtype:** create rigid subtypes that are the new direct parents of one or more anti-rigid subtypes. If only one rigid subtype is created, the modeler can optionally set it as derived by negation of the other rigid subtypes. The following OCL template is proposed to achieve that:

```
context NewRigid::allInstances() : Set(NewRigid)
derive : RigidParent.allInstances() -> select( x |
  not(x.oclIsTypeOf(Rigid1) or x.oclsIsTypeOf(Rigid2) or ... or
  x.oclIsTypeOf(Rigidn))
```

Anti-Pattern Relations

Group by Feature (Gen. Set): none

Group by Feature (Hierarchy): Declnt, Mixlden, MixRig, UndefPhase

Group by Type (Classification): DepPhase, HetColl, HomoFunc, Mixlden, MixRig, RelRig, UndefFormal, UndefPhase

Group by Type (Scope): DepPhase, FreeRole, ImpAbs, MultDep, HomoFunc, Mixlden, MixRig, RelRig, UndefPhase

Causes: none

Caused by: none

5.7 HETEROGENEOUS COLLECTIVE (HETCOLL)

As discussed in (GUIZZARDI, 2011), a collective is an entity whose parts (members) play the same role regarding it. If we say that a troupe is a collection of artists, we are implying that all artists just play the role of being part of the troupe. Conversely, functional complexes are entities whose parts play different roles regarding it. The CPU is a functional part of a computer, as well as the hard-drive, since the former is responsible for processing operations, whilst the latter for storing non-volatile data.

The differentiation proposed on the meta-conceptualization does not imply that all modelers will characterize a particular entity type, like computer or troupe, in the same way. In fact, it is quite ordinary for the opposite to happen: intuitively assumed to be complexes characterized as collections and vice-versa. We can define the “Computer” concept as a collection, if we assume that all its parts only play the role of being part of a computer. In the same way, one can understand the troupe concept as a functional complex, if one assumes that it contains actors, dancers and singers, and that they contribute differently to the function of the troupe.

UFO does not define collectives, however, only by their membership relations. They can also be refined into sub-collections. Note that these type of parts provide further structure to the collection, but do not differentiate roles played by their members. The troupe example, if modeled as a collective, could be refined into the singer, dancer and actors sub-collections, whose members are the artist who can sing, dance and act, respectively. The difference from functional complex view on troupe is that although segregated into sub-collections, all artists are still just members of the troupe.

The Heterogeneous Collective (HetColl) anti-pattern identifies collectives that are composed by two different types of members, which is an indication that the modeler might have confused the collection and functional complex concepts or the membership and sub-collection relations. The identification algorithm is to find a type that only allows collection instances and that is connected (directly or indirectly – though one of its ancestor types) to two or more memberOf relations in the whole end.

The characteristic of only allowing collection instances is true for a given type if:

- it is stereotyped as collective;
- if it is stereotyped as subkind, role or phase and is a direct or indirect sub-type of another type stereotyped as collective; or
- it is stereotyped as mixin, category or roleMixin and all its direct or indirect sortal children obey the last two conditions.

The key aspect to successfully analyze this anti-pattern is to identify the whole's perspective towards the parts. If one concludes that the parts in fact play different roles w.r.t the whole, the refactoring plan is to change the nature of the whole to functional complex (if necessary, also change the nature of parts) and change the stereotype of the meronymic relations to componentOf.

Setting the nature of a type t to collective, functional complex or quantity is a task that depends on the current stereotype of t . We exemplify it by explaining the process to change a type's nature to that of a collection.

- t is stereotyped as quantity or kind, change the stereotype to collective;

- t is stereotyped as subkind, role or phase, one can either change the stereotype of the current identity provider to collective, select another that is already stereotyped as collective or create a new one stereotyped as collective; and
- t is stereotyped as category, mixin or roleMixin, repeat the process defined in the two previous items for every sortal subtype.

During the analysis of the anti-pattern, however, one can conclude that the members indeed play the same role regarding the whole. In those cases, the proposed refactoring is to make this conclusion explicit by creating a type as the direct parent of all current types and merging all memberOf relations into one, which is connected to the new super type.

A modeler should take the last alternative when she concludes that the member types are in fact sub-collections, i.e., they are refinements of internal structure of the collective whole. To achieve this desired conceptualization, one must change the stereotypes of the memberOf relations to subCollectionOf and, if necessary, change the nature of the part types to collection.

Table 37 summarizes the structure, identification and refactoring alternatives proposed for the *HetColl* anti-pattern.

Table 37. Characterization of the *HetColl* anti-pattern.

Name (Acronym)		Description
Heterogeneous Collective (<i>HetColl</i>)		A collection type connected to two or more different member parts through «memberOf» relations.
Type	Feature	Justification
Classification	Part-Whole	The multiple part types, the main characteristic of this anti-pattern, indicate that the modeler might have confused the concepts of collection and functional complex or the different relations of membership and sub-collection.
Pattern Roles		
Mult.	Name	Possible Types
1	Whole	«collective», «subkind», «phase», «role», «category», «roleMixin» and «mixin»
2..*	partOf-n	«memberOf»
2..*	Part-n	«kind», «collective», «subkind», «phase», «role», «category», «roleMixin» and «mixin»,

Additional Constraints

1. Only collections may instantiate the Whole
2. Only collections and functional complexes may instantiate all Part-n
3. Let M be the set of memberOf relations identified in an HetColl occurrence, w the class identified as the Whole, wholeType(r) the function that return the class connected to the whole end of a meronymic relation r, and ancestorSet(c) the function that returns all direct and indirect super types of a class c:

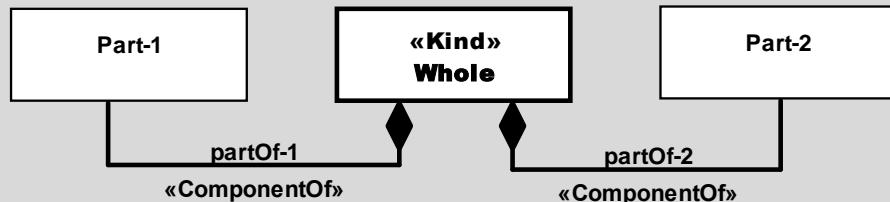
$$\forall m \in M, \text{wholeType}(m) = w \vee \text{wholeType}(m) \in \text{ancestorSet}(w)$$

Generic Example

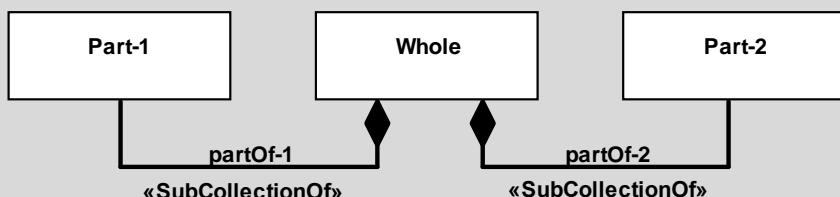


Refactoring Plans

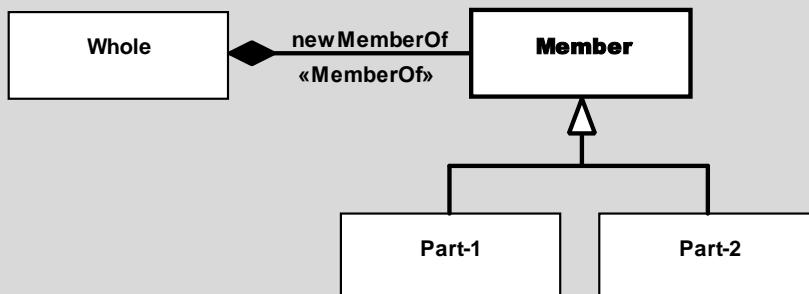
1. **[Mod] Set as functional parts:** Change the collection nature of Whole to functional complex and change the stereotype of the partOf relations to «componentOf». If the part types are also not exclusively functional complexes, fix them to.



2. **[Mod] Set as sub-collections:** Change the stereotype of the partOf relations to «subCollectionOf». If the part types are also not exclusively collections, enforce it.



3. **[Mod/Del] Set generic membership:** Create a common direct parent type for all part types, remove all existing partOf relations and create a new one from the whole to the created parent. The stereotype of the parent is derived from the stereotype of the parts.



Anti-Pattern Relations

Group by Feature (Part-Whole): HomoFunc, WholeOver, PartOver

Group by Type (Classification): DepPhase, GSRig, HomoFunc, MixIden, MixRig, RelRig, UndefFormal, UndefPhase

Causes: ImpAbs (3)

Caused by: none

We use as a *HetColl* occurrence example a fragment extracted from the IDAF model, which describes the domain of the Institute of Agricultural Protection, a governmental organization. Figure 30 presents a fragment that describes a particular type of work group, named Technical Administrative Support Group, which has employees that play the roles of being technical and/or administrative support. The cardinalities constraints defined in the part ends show that this type of work group requires employees performing different duties. If they play different roles, the work group should be a functional complex and not a collective and, thus, characterize a modeling error.

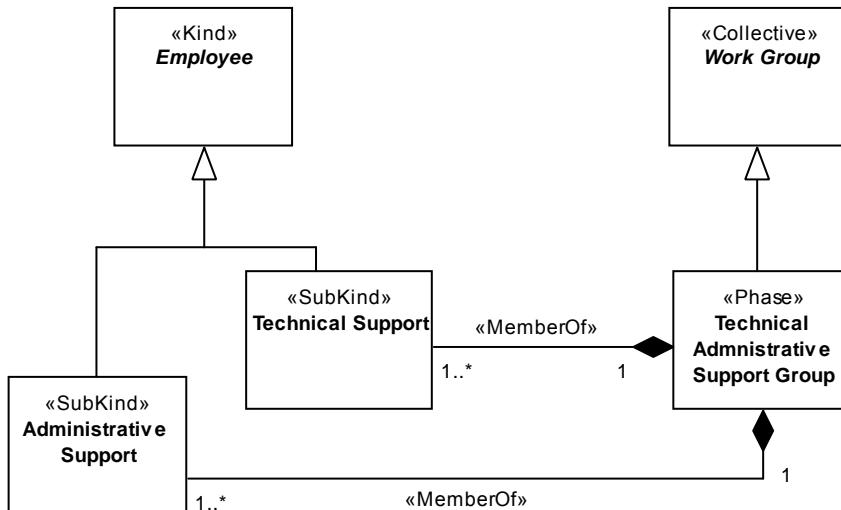


Figure 30. Simplified fragment of the IDAF model that depicts the *HetColl* anti-pattern.

Although the focus of the anti-patterns is not on the reason that leads modelers to make the error-prone decisions, we make an exception for *HetColl*. Our empirical experience in analyzing ontologies and discussing them with their respective authors indicates that whenever a collective noun (like fleet, group, pack) is used, modelers are most likely to represent it as a collective, without even analyzing the context of the particular conceptualization they are formalizing.

5.8 HOMOGENEOUS FUNCTIONAL COMPLEX (HOMOFUNC)

The Homogeneous Functional Complex (HomoFunc) is the counter part of the *HetColl* anti-pattern. They are motivated for the same reasons but, in this case, functional complexes are under analysis. The identification structure is quite simple, a class that can only have functional complex instances directly connected to exactly one componentOf relation and indirectly connected to none.

A type only allows instances of functional complex if:

- it is stereotyped as kind;
- if it is stereotyped as subkind, role or phase and is a direct or indirect sub-type of another type stereotyped as kind;
- it is stereotyped as mixin, category or roleMixin and all its direct or indirect sortal children obey the last two conditions.

As discussed in Chapter 2 and in the *HetColl anti-pattern*, functional complex have heterogeneous structures. That means having different types of part contributing in varied ways to the functionality of the whole. This anti-pattern investigates an allegedly homogenous structure of a functional complex.

The first refactoring plan is to transform the functional part-hood in a membership. Change the nature of the whole to collection and the stereotype of the relation to memberOf. Modeler should execute this prescribed action if they intended to represent the homogeneous structure.

Conversely, if a heterogeneous structure is intentional, the modeler should specify additional types of parts. That can be achieve in two different, but non-exclusive ways. First, through the creation of subtypes of the single functional part, alongside with extra componentOf relations. Second, through the specification of new functional parts, which do not related to the existing one.

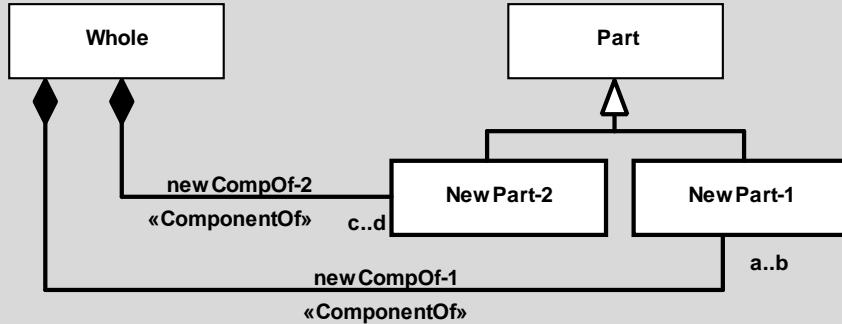
Note that specifying functional part-hood relations to overlapping subtypes characterize an occurrence of the *WholeOver* anti-pattern (see the definition in Section 5.21). If that happens, the modeler should proceed to check this new occurrence after completing the current analysis.

Table 38 summarizes the description of the *HomoFunc* anti-pattern.

Table 38. Characterization summary of the *HomoFunc* anti-pattern.

Name (Acronym)		Description
Homogeneous Functional Complex (<i>HomoFunc</i>)		A functional complex type connected to a single part through a «componentOf» relation.
Type	Feature	Justification
Classification; Scope		
Part-Whole		If a whole is composed by a unique type of part, it is most likely that all of the part's instances play the same role w.r.t. their whole. That homogeneous structure is not a characteristic of a functional complex.
Pattern Roles		
Mult.	Name	Possible Types
1	Whole	«kind», «subkind», «phase», «role», «category», «roleMixin» and «Mixin»
1	Part	«kind», «subkind», «phase», «role», «category», «roleMixin» and «Mixin»
1	partOf	«componentOf»
Additional Constraints		
<ol style="list-style-type: none"> Only functional complexes may instantiate the Whole Only functional complexes may instantiate the Part Whole is not indirectly connected, at the whole end, to any componentOf. partOf's lower bound multiplicity of the part end must be greater or equal to 2 		
Generic Example		
<pre> graph LR Whole[Whole] -- "partOf a..b «ComponentOf»" --> Part[Part] </pre>		
Refactoring Plans		
<ol style="list-style-type: none"> [Mod] Set as membership: Change the functional nature of Whole to and change the stereotype of the «componentOf» to «memberOf». 		
<pre> graph LR CollectiveWhole["«Collective»\nWhole"] -- "partOf a..b «MemberOf»" --> Part[Part] </pre>		
<ol style="list-style-type: none"> [New] Add functional parts: Create one or more functional parts for Whole. 		
<pre> graph TD Whole1[Whole] -- "partOf a..b «ComponentOf»" --> Part[Part] Whole2[Whole] -- "new CompOf c..d «ComponentOf»" --> NewPart[New Part] </pre>		

3. **[New] Add part subtypes***: Create one or more subtypes of Part and connected them to Whole through exclusive «componentOf» relations. The original relation might be kept, but if so, the new relations must subset, redefine or specialize it.



* Adopting this solution generates an occurrence of the *WholeOver* anti-pattern.

Anti-Pattern Relations

Group by Feature (Part-Whole): HetColl, WholeOver, PartOver

Group by Type (Classification): DepPhase, GSRig, HetColl, MixIden, MixRig, RelRig, UndefFormal, UndefPhase

Causes: WholeOver (3)

Caused by: none

Figure 31 brings an *HomoFunc* example extracted from the PAS 77 ontology. This model fragment formalizes a conceptualization regarding information technology architecture. Notice that the IT Architecture class is defined solely as composed by IT Components, which in turn can be sites, platforms, operating systems and data storage units. As is, the model treats all architectural parts in the same way, as being a component. If that is the case, the most appropriate solution would be to represent IT Architecture as a collective of IT Components.

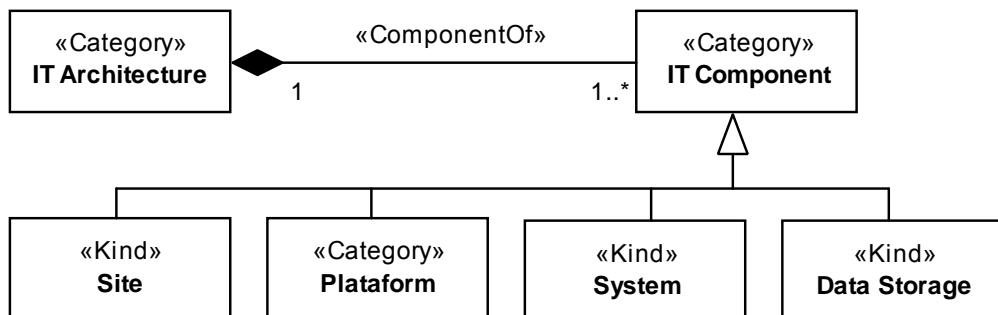


Figure 31. *HomoFunc* occurrence encountered in the PAS 77:2006 ontology.

5.9 IMPRECISE ABSTRACTION (IMPABS)

An association R characterizes the logical anti-pattern named Imprecise Abstraction (ImpAbs) if at least one of the following holds:

- R's source end upper bound multiplicity is equal or greater than 2 and the Class connected to it has 2 or more subtypes;
- (ii) R's target end upper bound multiplicity is equal or greater than 2 and the Class connected to it has 2 or more subtypes

ImpAbs indicates structures that can be too permissive, i.e. allow undesired model instantiations. Representing a generic relation (between super-types of a hierarchy) causes one to “lose control” on how many instances of a particular subtype an instance of the opposite type may be connected to. Furthermore, it precludes the specification of other particular meta-property values for the association (like *isDerived*, *isReadOnly*, *isEssential* and *isInseparable*) when it connects individuals that are also instances of subtypes of the related types.

ImpAbs provides three refactoring alternatives:

- (a) set cardinality constraints through the specification of an OCL invariant;
- (b) set cardinality constraints through the specification of a new association that subsets the original one; and
- (c) the specification of particular association meta-property values, also through the creation of an association.

Options (a) and (b) are equivalent in term of logical implications and, thus, are mutually exclusive for the same pair of classes. Conversely, alternative (c) is combinable with the first two, although if one is already going to create a new association, it is more reasonable to use it also to set the cardinality constraints.

Notice that the constraints defined for relations to/between subtypes of the originally related classes (using our example, the definition of a relation from Heart to Pacemaker Cell) cannot contradict the ones for the original relation. Minimum cardinalities must be lower or equal to the general's relation minimum and maximum. Maximum cardinalities must be greater than the general's relation minimum and lower or equal to the general's

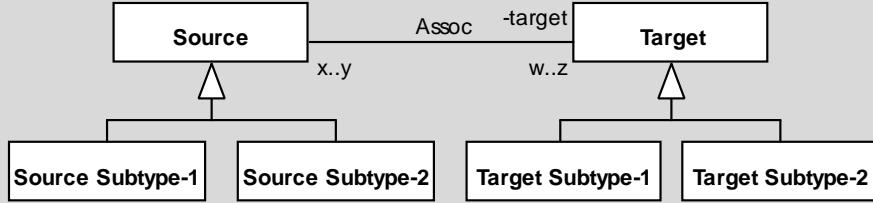
maximum. Modelers can only customize other Boolean meta-properties, like *isEssential*, *isInseparable*, *isImmutableWhole*, *isImmutablePart*, *isShareable* or *isReadOnly* if the value set for the generic relation is false.

Table 39 resumes the characterization of the ImpAbs anti-pattern.

Table 39. Characterization summary of the ImpAbs anti-pattern.

Name (Acronym)		Description
Imprecise Abstraction (ImpAbs)		A given association R characterizes an ImpAbs occurrence if at least one of the following holds: (i) R's source end upper bound multiplicity is equal or greater than 2 and the Class connected to it has 2 or more subtypes; (ii) R's target end upper bound multiplicity is equal or greater than 2 and the Class connected to it has 2 or more subtypes
Type	Feature	Justification
Logical; Scope		
Association		Representing a general relation occasionally causes the model to be too permissive because one “loses control” on how many instances of a particular subtype an instance of the opposite type may be connected to. Furthermore, it precludes the specification of other particular meta-property values, like <i>isDerived</i> and <i>isReadOnly</i> for all associations, and <i>isEssential</i> and <i>isInseparable</i> for meronymics.
Pattern Roles		
Mult.	Name	Possible Types
1	Assoc	All association stereotypes
1	Source	All class stereotypes
1	Target	All class stereotypes
0..*	Source Subtype-n	All class stereotypes
0..*	Target Subtype-n	All class stereotypes
Additional Constraints		
<ol style="list-style-type: none"> Let <i>allSubtypes(c)</i> be the function that return all direct and indirect subtypes of a class c, <i>sourceEnd(a)</i> and <i>targetEnd(a)</i> the functions that return the source and target ends of an association a, and <i>upper(p)</i> be the function that return the upper bound cardinality of a property p, then: $(upper(sourceEnd(Assoc)) \geq 2 \wedge \#allSubtypes(Source) \geq 2) \vee (upper(targetEnd(Assoc)) \geq 2 \wedge \#allSubtypes(Target) \geq 2)$ Let <i>SoChildren</i> be the set of all classes identified as Source Subtype-n, then: $\forall x \in SoChildren \mid x \in allSubtypes(Source)$ Let <i>TgChildren</i> be the set of all classes identified as Target Subtype-n, then: $\forall x \in TgChildren \mid x \in allSubtypes(Target)$ 		

Generic Example



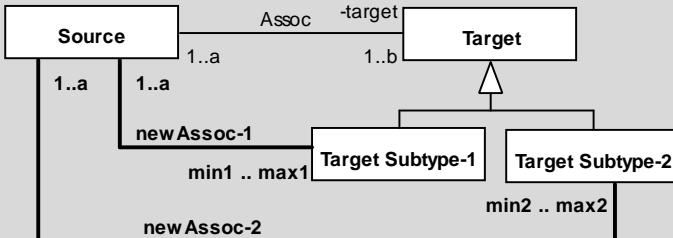
Refactoring Plans

- [OCL] Add multiplicity constraint:** choose this option if there is a domain restriction that requires an instance of Source, or of one of its subtypes, to be connected to a minimum, maximum or precise number of instances of Target, or one of its subtypes. The following OCL invariant enforces the desired constraint:

```

context Source
inv: let sub1Size = self.target->select( x |
xoclIsTypeOf(_'Target Subtype-1'))->size() in
sub1Size >= min1 and sub1Size <= max1
  
```

- [New] Add multiplicity constraint (subsetting association):** this option has the same logical result of the first one. However, the results are achieved through the specification of a new association (using the same stereotype of Assoc) that subsets Assoc and whose cardinalities enforce the cardinality constraints.



- [New] Add custom meta-property (subsetting association):** choose this option if the relation between Source and Target have particular meta-properties (like isReadOnly and isEssential) when an instance of Source, or of one of its subtypes, to be connected to a minimum, maximum or precise number of instances of Target, or one of its subtypes

Anti-Pattern Relations

Group by Feature (Association): AssCyc, BinOver, RelComp, RelSpec

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Group by Type (Scope): DepPhase, FreeRole, MultDep, GSRig, HomoFunc, Mixlden, MixRig, RelRig, UndefPhase

Causes: none

Caused by: HetColl

To exemplify, consider the occurrence identified in the Electrocardiogram (ECG) ontology and depicted in Figure 32. The fragment states that a Heart contains atriums, ventricles and cells. The componentOf between Heart and Heart Cell characterizes the

ImpAbs occurrence. According to the model, a Heart may contain only Non Pacemaker Cells. The catch is that the pacemaker cells are the ones responsible for the heart's contraction, i.e. a heart without them would not beat. To solve this problem, we would need to create additional relations, one for each type of required cell.

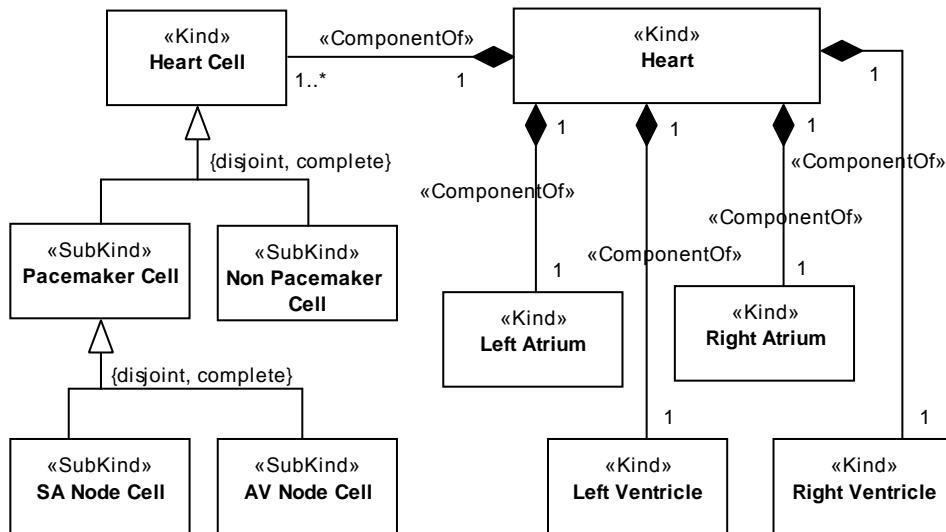


Figure 32. ImpAbs occurrence identified in the ECG Ontology.

As a last remark, we highlight that our empirical studies indicated that ImpAbs is more likely to be a problem when the association that characterizes the occurrence is a part-whole relation (with exception of memberOf), like in the provided example.

5.10 MIXIN WITH SAME IDENTITY (MIXIDEN)

In UFO's substantial hierarchy, the first classification regards the separation of sortals and non-sortals (or mixins, in the general sense). The former aggregates individuals that follow the same identity principle, whilst the latter encompasses individuals that follow different ones. If we take for example, an organization's VAT number (or in Brazil, an organization's CNPJ) as their identity principle, then the concepts of school, supermarket and organization would be sortals. Now, if we adopt a person's SSN number (or in Brazil, one's CPF) as their identity principle, the concept of Client, which encompasses both people and organizations, would be a mixin.

The Mixin With Same Identity (MixIden), as the name suggests, is motivated by the use of a mixin stereotype to qualify a type whose instances all obey the same identity principle. In structural terms, it corresponds to a class (that we now refer to as Non-Sortal) stereotyped as either «mixin», «roleMixin» or «category» specialized only by sortals classes that supply identity or share a common identity provider ancestor.

In order for all subtypes to follow the same identity principle, there may be at most one class stereotyped as «kind», «quantity» or «collective» as the subtype of Non Sortal. Nonetheless, there is no restriction to the number of «subkind», «role» and «phase» subtypes, as long as they are all direct or indirect subtypes of the same identity provider.

The analysis of this anti-pattern is quite simple. If the non-sortal type really does not allow individuals with other identity principles, then it is not a mixin. In this case, all that is required to fix the model is to change the stereotype of Non Sortal to a sortal one («subkind», «role» or «phase») and create a generalization from the Non Sortal class to the common identity provider.

Conversely, if the mixin class does allow different identity principles, the modeler should make them explicit. To do that, she can: (i) create generalizations from existing types to the mixin; or (ii) create new types and generalize them into mixin.

Table 40 presents the complete summary of the MixIden anti-pattern.

Table 40. Characterization summary of the *MixIden* anti-pattern.

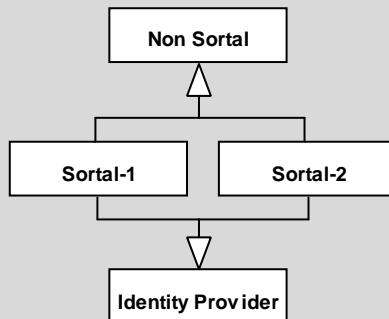
Name (Acronym)		Description
Mixin With Same Identity (MixIden)		A non-sortal class specialized only by sortal types that follow the same identity principle (by inheriting it or supplying it).
Type	Feature	Justification
Classification; Scope	Hierarchy; Mixin	The common characteristic of all different types of mixin classes is the aggregation of individuals that follow different identity principles. The reason to analyze this anti-pattern is that a non-sortal should not be specified as a sortal or it may convey the wrong meaning.
Pattern Roles		
Mult.	Name	Possible Types
1	Non Sortal	«mixin», «roleMixin» and «category»

1	Sortal-n	«subkind», «role», «phase», «kind», «quantity» and «collective»
1	Identity Provider	«kind», «quantity» and «collective»

Additional Constraints

1. For every Subtype-n, either one of the following holds: (i) Sortal-n = Identity Provider; or
(ii) Identity Provider is an ancestor of Sortal-n

Generic Example



Refactoring Plans

1. **[Mod/New] Change Mixin to Sortal:** change the stereotype of Mixin to either subkind, role or phase and create a generalization from Mixin to Identity Provider.
2. **[New] Add Sortal Subtypes:** add new or existing sortal sub-types to Mixin that do not follow the same identity principle of defined by Identity Provider.

Anti-Pattern Relations

Group by Feature (Hierarchy): GSRig, DeclInt, MixRig, UndefPhase

Group by Feature (Mixin): MixRig

Group by Type (Classification): DepPhase, GSRig, HetColl, HomoFunc, MixRig, RelRig, UndefFormal, UndefPhase

Group by Type (Scope): DepPhase, FreeRole, ImpAbs, MultDep, GSRig, HomoFunc, MixRig, RelRig, UndefPhase

Causes: GSRig (1)

Caused by: none

Now, we discuss a Mixlden example found in the MGIC ontology. Figure 33 depicts the formalization of a fragment of the infrastructure concession sub-domain. The class Concessionaire represents the companies to whom the government delegates the responsibility of maintaining and developing the road infrastructure in exchange for the right to charge drivers a toll. The class Public Organization encompasses all different public structures, like regulatory agencies, institutes, public-owned companies and so

on. The diagram, as a whole, captures that public organizations and concessionaries perform maintenance works on roadway components.

The RoleMixin class named “*Responsible for Maintenance*” and the subtypes “*PO Responsible for Maintenance*” and “*Concessionaire Responsible for Maintenance*”, characterize the Mixlden example, since both subtypes inherit their identities from the kind “*Organization*”. In this context, something other than an organization cannot be responsible for conducting maintenance works. Therefore, the authors should change the class “*Responsible for Maintenance*” to a simple role.

The improper use of the RoleMixin anti-pattern causes errors like the aforementioned one. They do not imply in consistency problems, but impairs the proper ontological classification of the type.

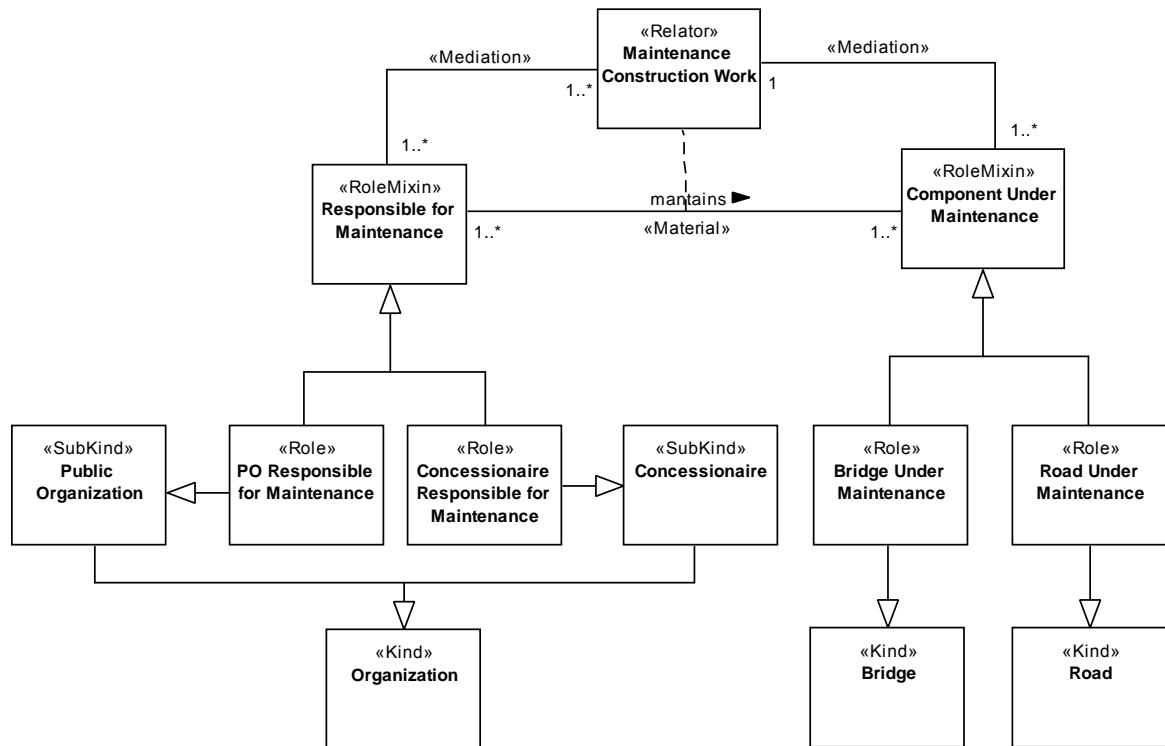


Figure 33. Simplified fragment of the MGIC that characterizes a Mixlden occurrence.

5.11 MIXIN WITH SAME RIGIDITY (MixRig)

The mixin stereotype is a particular type of non-sortal that not only aggregate individuals that follow different identity principles but also are semi-rigid. Semi-rigidity means that they type behaves as rigid for some of its individual and anti-rigid for others. Guizzardi (2005), provides the “Seatable” concept as an example of Mixin. This type accounts for all things on which people can sit. Now, if we consider the concepts of chair and crate, for example, we notice that the former must always be seatable, whilst the latter can eventually be.

The goal of this anti-pattern is to verify if the mixin stereotype is being properly used, i.e., to verify if it qualifies a type which is necessarily applicable for a group of individuals and optionally for others. This question is raised because an occurrence of the Mixin With Same Rigidity (MixRig) consists in a Mixin specialized in type that are either all rigid (stereotyped as «subkind», «kind», «quantity» or «collective») or all anti-rigid (stereotyped as «role», «phase» or «roleMixin»).

MixRig proposes three refactoring alternatives:

- enforce a particular rigidity value by changing the stereotype of the mixin type;
- fix the rigidity value of one of the sub-type by also changing its stereotype; and
- add new or existing classes as subtypes of mixin to properly characterize its semi-rigidity.

Table 41 summarizes the characterization of the MixRig anti-pattern.

Table 41. Characterization summary of the *MixRig* anti-pattern.

Name (Acronym)	Description	
Mixin With Same Rigidity (MixRig)	A class stereotyped as «mixin» specialized only by other classes that have the same rigidity property, i.e., are all rigid or all anti-rigid.	
Type	Feature	Justification
Classification; Scope	Hierarchy; Mixin	As all non-sortals, mixins aggregated individuals that follow different identity principles. Its distinguishing characteristic, though, is that is semi-rigid, i.e., it behaves as a rigid type for some individuals as an anti-rigid for others. This anti-pattern analyzes mixins that, despite their capabilities, only generalize types with the same rigidity.

Pattern Roles				
Mult.	Name	Possible Types		
1	Non Sortal	«mixin»		
1	Subtype-n	«subkind», «role», «phase», «kind», «quantity», «collective», «roleMixin» and «category»		
Additional Constraints				
1.	All sortals are rigid («subkind», «kind», «quantity», «collective» and «category») or all sortals are anti-rigid («role», «phase» or «roleMixin»)			
Generic Example				
<pre> classDiagram class Mixin { <<Mixin>> } class Subtype1 class Subtype2 class Subtype3 Mixin < -- Subtype1 Mixin < -- Subtype2 Mixin < -- Subtype3 </pre>				
Refactoring Plans				
1.	[conditional] [Mod] Change mixin to category: if all subtypes are rigid, and no anti-rigid subtype is expected to specialize Mixin, change the stereotype to «category».			
2.	[conditional] [Mod] Change mixin to roleMixin: if all subtypes are anti-rigid, and no rigid subtype is expected to specialize Mixin, change the stereotype to «roleMixin».			
3.	[Mod] Change subtypes stereotypes: this solution is a recognition that the semi-rigidity of Mixin is correct and consists in changing the stereotype of one or more subtypes of Mixin to properly characterize the semi-rigidity.			
4.	[New/Mod] Add subtypes: set new or existing types as direct children of Mixin in order to properly characterize the semi-rigidity.			
Anti-Pattern Relations				
Group by Feature (Hierarchy): GSRig, DeclInt, Mixlden, UndefPhase Group by Feature (Mixin): Mixlden Group by Type (Classification): DepPhase, GSRig, HetColl, HomoFunc, Mixlden, RelRig, UndefFormal, UndefPhase Group by Type (Scope): DepPhase, FreeRole, ImpAbs, MultDep, GSRig, HomoFunc, Mixlden, RelRig, UndefPhase Causes: none Caused by: none				

Figure 34 depicts an occurrence of the MixRig anti-pattern identified in the MGIC ontology. The fragment describes the different types of railway properties, every building or lot that composes the required infrastructure to provide rail transportation services. Examples of railway buildings are terminals (also referred to as stations) and yards. The first is the place where people get in and out of trains and the second, a series of rail tracks to load and unload cargo, and for sorting and storing railroad cars. Railway lots are properties that composed the infrastructure but have no buildings in it, like a Railroad Right-of-Way, the areas alongside railroads reserved for transportation purposes.

The problem addressed by the anti-pattern is that the modelers formalized the concept of “*Railway Property*” as a Mixin, however, all its subtypes are rigid (“*Railway Building*” and “*Railway Lot*”). Through the interaction with the modelers, we reached the conclusion that this was indeed an error because being a *Railway Lot* is not a necessary condition, i.e., it should be a role of lot instead of a subkind.

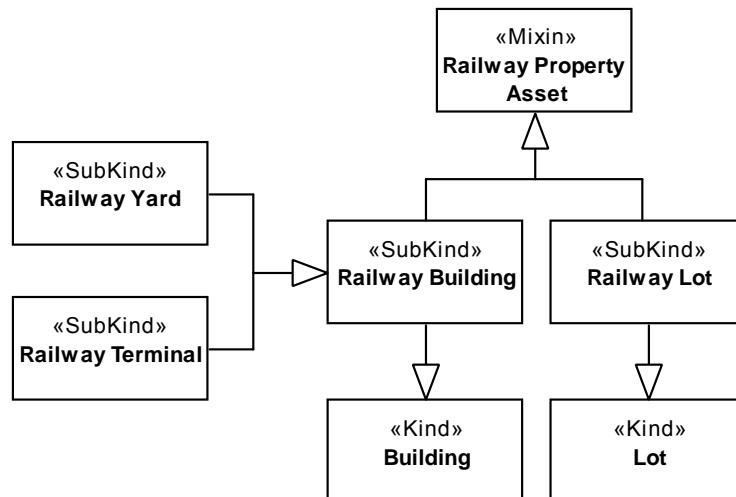


Figure 34. Simplified MixRig occurrence identified in the MGIC ontology

5.12 MULTIPLE RELATIONAL DEPENDENCY (MULTDEP)

Roles are externally relationally dependent types. This ontological property implies that the instantiation of roles derives from the establishment of particular relations to other individuals. OntoUML, however, does not limit the formalization of relational dependencies to classes stereotyped as role or roleMixin. In fact, a modeler might

connect a mediation (the characterization of relational dependency) to any class with one of the following stereotype: kind, collective, quantity, subkind, role, phase, category, mixin and roleMixin.

The *Multiple Relational Dependency (MultDep)* anti-pattern aims to investigate a particular subset of externally dependent types: the ones that require more than one dependency simultaneously. Structurally, its identification consists of an object class (stereotyped as «kind», «quantity», «collective», «role», «phase», «subkind», «category», «mixin» or «roleMixin») directly connected to two or more mediations.

The first step to analyze this anti-pattern is to verify if all dependencies are definitely mandatory. If the answer is no, they should be transformed to optional dependencies through the creation of one role per optional dependency.

If a modeler concludes that two or more dependencies are optional for a given type, she should also check if they are established in particular order. To exemplify, consider the following optional relational dependencies of a person; a dependency that holds between a person and her employer; and another dependency, between a person and the school in which she studies. *A priori*, one cannot assert a particular order for the instantiation of these dependencies, since a person can become a student and later become an employee or the other way around. Conversely, if we also consider the relational dependency between an internship and a student, we clearly identify an order. For a person to become an intern, she needs to be a student already.

If the class that characterizes the MultDep occurrence has more than two mediations connected directly to it, a modeler might refactor it by simultaneously defining ordered and unordered optional dependencies. Nonetheless, when creating unordered dependencies, all new roles should specialize the same super-type. If an order is required, the modeler should create the new roles as subtypes of one another, in a sort of “hierarchy line”.

Regardless if a MultDep occurrence identifies optional dependencies or not, the next step is to analyze if there are “dependencies between dependencies”, i.e., a relator that formalizes a dependency is somehow related to a relator that formalizes another dependency. To simplify, consider the following short story:

A person becomes an undergraduate student when she enrolls in a major course at a university, e.g. “Computer Science” or “Philosophy”. A unique number identifies each enrollment. Victor, a very curious and dedicated young man, decides to pursue, simultaneously, a major in “Philosophy” and “Computer Science”. To do that, he would need to enroll two times at the university. After his enrollments, Victor wants to apply for “Logics 101” as a “Computer Science” student and apply for “Sociology 101” as a “Philosophy” major. To do that, each course application must not only identify “Victor” as the applying student, but also identify the particular enrollment he is using to apply.

The identification of the enrollment in the course application characterizes the dependency between the relators “Major Enrollment” and “Course Enrollment”. We propose the formalization of this relation in an OntoUML model as a formal association.

Representing the dependency between two relators using a formal association potentially generates two anti-pattern occurrences: one AssCyc (see Section 5.1) and one UndefFormal (see Section 5.19). The former requires attention and the modeler should properly analyze it. The latter, conversely, is just a “false alarm”.

We summarize the description of the MultDep anti-pattern in Table 42.

Table 42. Characterization summary of the MultDep anti-pattern.

Name (Acronym)		Description
Multiple Relational Dependency (MultDep)		An object class directly connected to two distinct «relator» types through «mediation» associations. The relators may not be direct or indirect specializations of one another.
Type	Feature	Justification
Logical; Scope	Relator	Externally dependent types, like all roles, require one dependency to characterize them. Whenever more than one is provided, it can indicate redundancy, scope issues and/or modeling an extra relation between the relators that characterize the dependency
Pattern Roles		
Mult.	Name	Possible Types
1	Type	«kind», «quantity», «collective», «role», «phase», «subkind», «category», «mixin» and «roleMixin»
2..*	Med-n	«mediation»
2..*	Relator-n	«relator»

Additional Constraints

- Let R be the set of all Relator in a MultDep occurrence and $isAncestor(c1,c2)$ the binary predicate that returns true if class $c1$ is a direct or indirect super-type of class $(c2,c1)$:

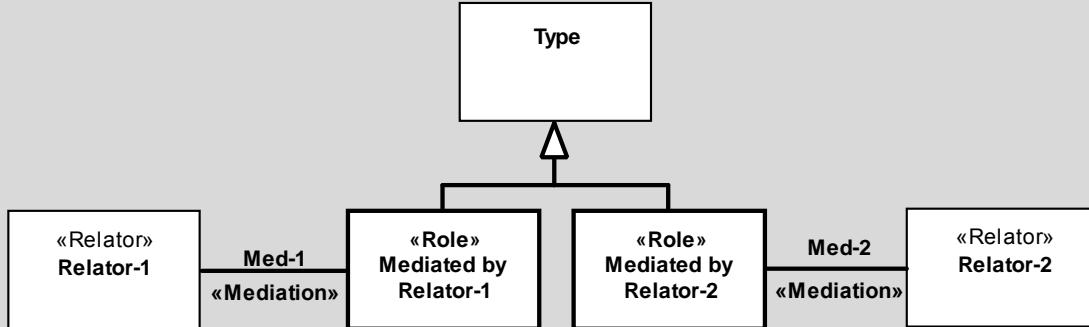
$$\forall r1, r2 \in R, \neg isAncestor(r1, r2) \wedge \neg isAncestor(r2, r1)$$

Generic Example

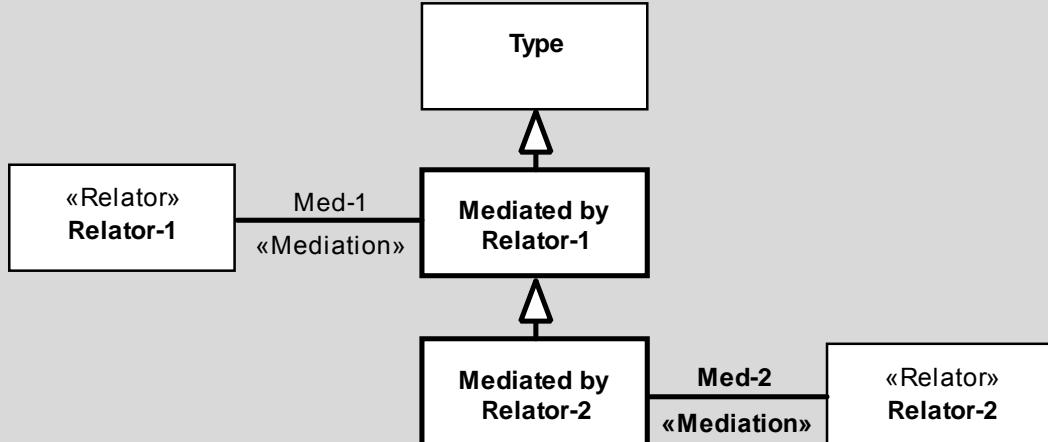


Refactoring Plans

- [New/Mod] Unordered optional dependencies:** Create a direct subtype of Type for each dependency. (In the example below, all dependencies were set as optional for Type)



- [New/Mod] Ordered optional dependencies:** Create a hierarchy line for dependencies, which an instance of Type can only acquire after others. (In the example below, all dependencies were set as optional for Type)



- [New] Create dependency between relators:** Create formal relations connecting relators that depend on one another. This solution generates an occurrence of AssCyc (which the user should be analyzed) and an occurrence of UndefFormal (which the user can ignore).

Anti-Pattern Relations

Group by Feature (Relator): DepPhase, FreeRole, RelOver, RelRig, RepRel

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Group by Type (Scope): DepPhase, FreeRole, GSRig, ImpAbs, HomoFunc, Mixlden, MixRig, RelRig, UndefPhase

Causes: UndefFormal (3), AssCyc (3)

Caused by: none

We exemplify the MultDep anti-pattern with an occurrence encountered in the OntoEmerge ontology (FERREIRA, 2013). The model fragment depicted in Figure 35 describes some of the relevant properties of installations in the context of emergency plan generation. The fragment presents four relational dependencies for the kind “*Installation*”:

- a dependency to define the installation’s owner, either a person or company;
- another to specify how businessmen explore installations, like as a school, or a supermarket, to sell cars, amongst others;
- a third that formalizes the installation’s evacuation place – the place where people are taken in case of emergencies, like a fire or an earthquake;
- lastly, the relation that characterize the place where people inside the installation were taken in previous emergencies.

Now we analyze each dependency exclusively. The dependency captured by the material association “*is evacuated to*” is clearly optional, as proven by its multiplicities (zero or more on both ends). In OntoUML, optional cardinalities are discouraged in general, but in this particular case, they the syntax forbids them. Thus, to keep the optionality of have been evacuated, a role “*Evacuated Installation*” should be created. Secondly, we consider the “*has default*” dependency. Our common sense assumes that not every building has a formally defined evacuation place, even if law requires that. Nonetheless, these type of buildings might be out of the scope of the ontology, so we cannot assume that it is an optional dependency, even if it seems to be. The third dependency is the ownership, which is undoubtedly mandatory. Lastly, we assume as optional, the dependency regarding business exploration of an installation. We can think of many examples of installations used only for residential or governmental

purposes. Furthermore, we can even assume the possibility of abandoned installations.

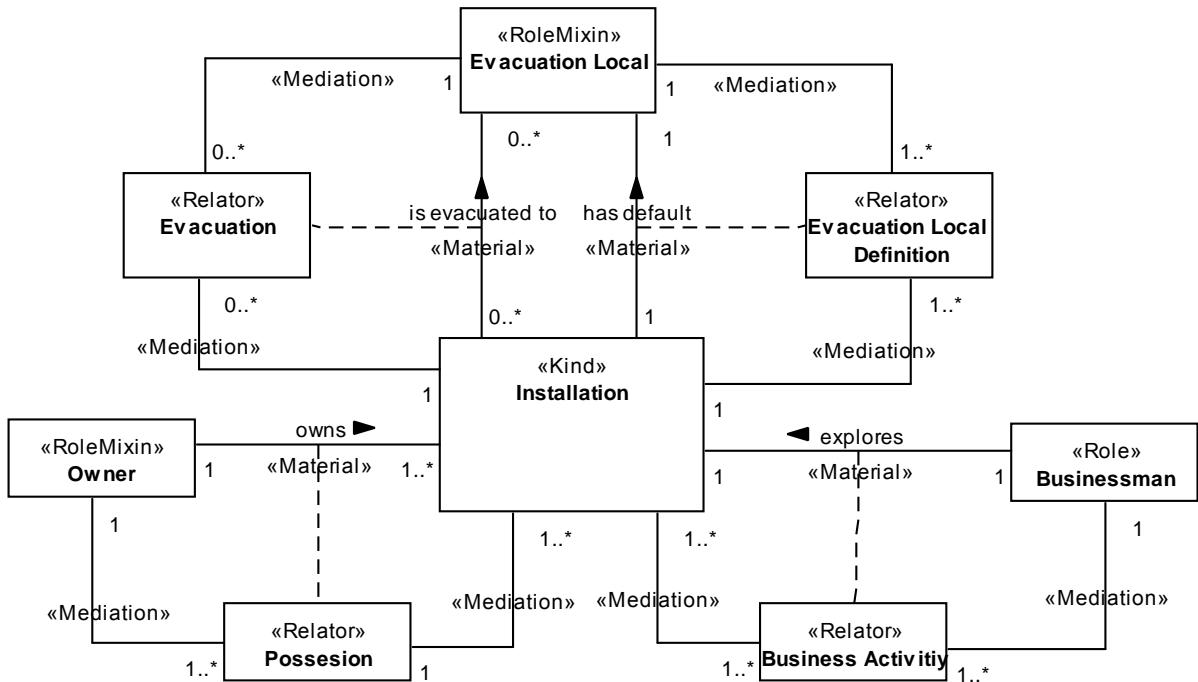


Figure 35. MultDep example extracted from the OntoEmerge ontology.

5.13 PART COMPOSING OVERLAPPING WHOLES (PARTOVER)

The Part Composing Overlapping Wholes (PartOver) anti-pattern follows the exact same logic as RelOver. The main difference is that, instead of the focus being on a relator mediating overlapping types, it is on a part composing overlapping wholes. We refrain from providing redundant definitions and just present a PartOver occurrence as an example. For more details, please refer to RelOver definition in Section 0.

A PartOver occurrence is depicted in Figure 36, an excerpt of the MGIC ontology (BASTOS et al., 2011). The part type “Watercourse”, that composes two different types of “Geographical Layer”, characterizes the PartOver occurrence. The overlapping wholes are the classes labelled as “Hydrographic” and “Drainage”. In fact, this fragment is part of a larger subdomain modelled to characterize the information managed by a logistic system.

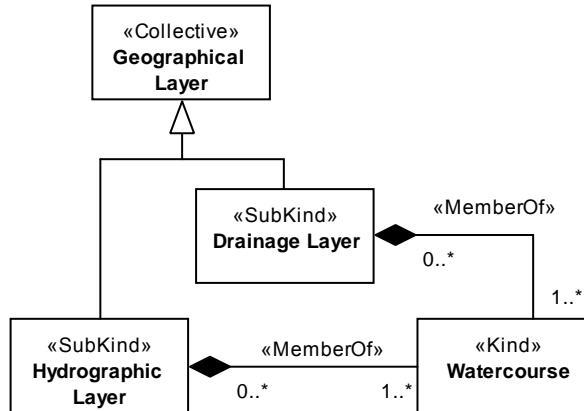


Figure 36. PartOver occurrence identified in the MGIC ontology.

The analysis starts by confirming the overlapping condition of Drainage and Hydrographic Layers. The feedback provided by the authors indicated that it was a false condition. These layers types should be disjoint. The adopted solution was to create a generalization set with the *isDisjoint* meta-attribute set to true. For the sake of completeness, if the layers were actually overlapping, the analysis would continue and the next step would be to verify whole exclusiveness. To enforce it, means to forbid the same whole to contain the same part through different relations. To achieve it, one should enrich the model with the OCL invariant defined in Listing 11.

Listing 6. OCL invariant generated to enforce exclusive wholes.

```

context Watercourse
inv: self.dreinage.oclAsType(_'Geographical Layer')->asSet()->excludesAll(
  self.hidrographic.oclAsType(_'Geographical Layer')->asSet())
  
```

Table 43 summarizes the structural pattern, the additional constraints and the refactoring plans for the PartOver anti-pattern.

Table 43. Characterization summary of PartOver the anti-pattern.

Name (Acronym)	Description	
Part Composing Overlapping Wholes (PartOver)	A part composing two or more whole types whose extension overlap. The sum of the meronymics' upper bound cardinalities of the whole end must be greater or equal to 2 or at least one of them be unlimited.	
Type	Feature	Justification
Logical	Part-Whole	This structure is usually too permissive. It is often the case that some of the whole types should be disjoint or set as exclusive in the context of a single part instance.

Pattern Roles		
Mult.	Name	Possible Types
1	Part	«kind», «collective», «quantity», «subkind», «phase», «role», «roleMixin», «category» and «mixin»
2..*	partOf-n	«subQuantityOf», «componentOf», «memberOf», «subCollectionOf»
2..*	Whole-n	«kind», «collective», «quantity», «subkind», «phase», «role», «roleMixin», «category» and «mixin»

Additional Constraints

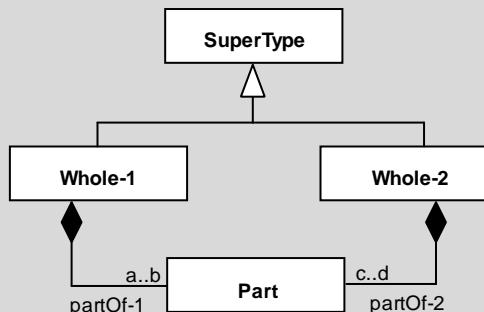
- Let M be the set of identified meronymic relations, wholeEnd(m) the function that returns the association end connected to the whole of a meronymic relation m, and upper(p) the function that return the upper bound cardinality of a property p, then:

$$\left(\sum_{m \in M} \text{upper}(\text{wholeEnd}(m_n)) \right) \geq 2$$

- Let O be the set of whole types that Part composes, then:

$$\exists x, y \in O \mid \text{overlap}(x, y)$$

Generic Example*



*Note: the presented structure is illustrative and do not cover all possibilities for PartOver occurrence

Refactoring Plans

- [OCL] Exclusiveness:** choose this option to forbid the same individual to play multiple roles w.r.t the same part instance. Create an OCL invariant according to the template:
`context Part
inv: self.whole1.oclAsType(Supertype) ->asSet() ->excludesAll(
self.whole2.oclAsType(Agent) ->asSet()) and`
- [OCL] Partially exclusiveness:** choose this option to set a subset of the whole types as exclusive.
- [Mod/New] Disjoint whole:** Enforce whole types to be disjoint through the creation or alteration of a disjoint generalization set.

Anti-Pattern Relations

Group by Feature (Part-Whole): HetColl, HomoFunc, WholeOver

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, WholeOver, RelOver, RelComp, RelSpec, RepRel

Causes: none

Caused by: none

5.14 RELATION COMPOSITION (RELCOMP)

The Relation Composition Anti-pattern (RelComp) is strictly logical. Its structural definition consists of two distinct associations, A and B, which connect ASource to ATarget and BSource to BTTarget, respectively. Furthermore, one of the following conditions must be true:

- BSource equals or is a subtype of ATarget and BTTarget equals or is a subtype of ATarget.
- BSource equals or is a subtype of ASource and BTTarget equals or is a subtype of ASource.

Figure 37 depicts is an excerpt of the ontological analysis performed on the Conceptual Scheme of the Human Genome (FERRANDIS; LÓPEZ; GUIZZARDI, 2013). It shows that an allele, an alternative form of a gene, is an ordered composition of nucleotides, organic molecules that are DNA's building blocks. The formal relation named "precedes" captures intended order.



Figure 37. Fragment of the CSHG ontology characterizing a RelComp.

Figure 38 illustrates a possible instantiation of the model excerpt. Notice the two distinct types of Allele: on the left, the precedence relation is only instantiated between nucleotides that compose the same allele; conversely, on the right, nucleotides precede others that composed different alleles. For this domain, the only allowed instantiations are the ones like the depicted on the left side of Figure 38.

We define five different constraints to determine how the instantiation of the relation "B" depends the instantiation of relation "A". In the genome example, how the "precedes" association depends on the "formedBy" association,

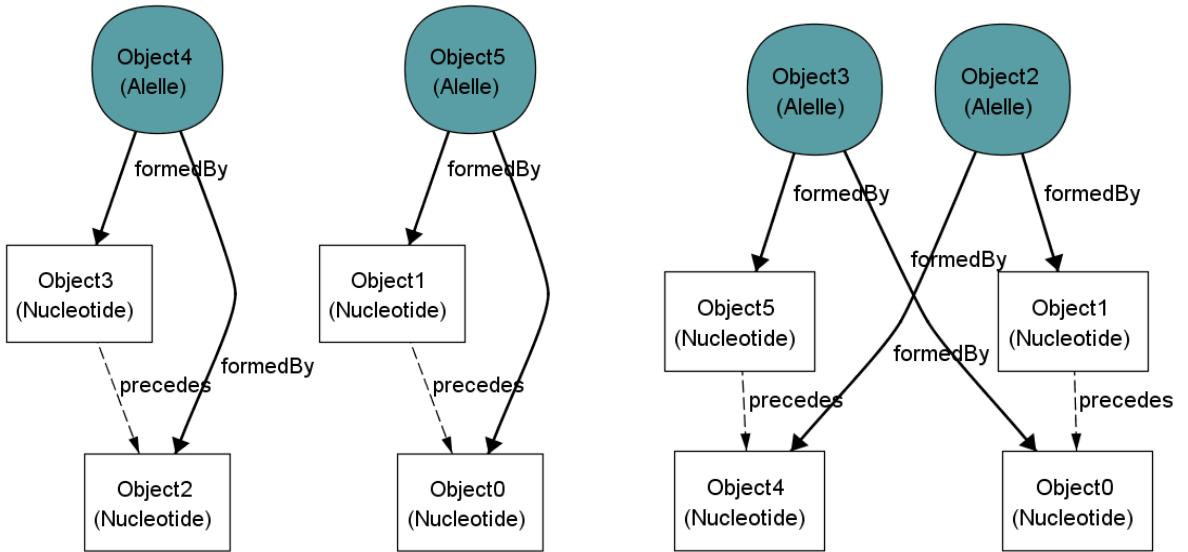


Figure 38. Generated examples of the CSHG excerpt. On the left, an expected instantiation. On the right, an undesired one.

Definition (Existential Composition): for every distinct individuals x, y , if x is related to y through B , it implies that x and y are related to at least one common individual through relation A . The formal definition, along with the respective OCL template, follows:

$$\forall x, y \mid B(x, y) \rightarrow \exists z \mid A(z, x) \wedge A(z, y)$$

Listing 7. OCL invariant to enforce existential composition.

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()->exists( z |
z.aTarget->asSet()->contains(self) and z.aTarget->asSet()->contains(y) )

```

Definition (Right Universal Composition): for every distinct individuals x, y , if x is related to y through B , it implies that all z that is connected to x , through A , is also connected to y , through A . The formal definition, along with the respective OCL template, follows:

$$\forall x, y \mid B(x, y) \rightarrow \forall z \mid A(z, x) \rightarrow A(z, y)$$

Listing 8. OCL invariant characterizing the Right Universal Composition.

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()->forAll( z |
z.aTarget->asSet()->contains(self) implies z.aTarget->asSet()->contains(y) )

```

Definition (Left Universal Composition): for every distinct individuals x, y , if x is related to y through B , it implies that all z that is connected to y , through A , is also connected to x , through A . The formal definition, along with the respective OCL template, follows:

$$\forall x, y | B(x, y) \rightarrow \forall z | A(z, y) \rightarrow A(z, x)$$

Listing 9. OCL invariant characterizing the Left Universal Composition.

```
context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()->forAll( z |
z.aTarget->asSet()->contains(y) implies z.aTarget->asSet()->contains(self))
```

Definition (Forbidden Composition): for every two distinct individuals x, y , if x is related to y through B , it implies that x and y are connected to no individual in common through A . The formal definition, along with the respective OCL template, follows:

$$\forall x, y | B(x, y) \rightarrow \nexists z | A(z, x) \wedge A(z, y)$$

Listing 10. OCL invariant that characterizes Forbidden Composition.

```
context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()->forAll( z |
not(z.aTarget->asSet()->contains(y) and z.aTarget->asSet()-
>contains(self)))
```

Definition (Custom Existential Composition): for every distinct individuals x, y , if x is related to y through B , it implies that x and y are connected to [less than / more than / exactly] n common individuals through A . The formal definition, along with the respective OCL template, follows:

$$\forall x, y | B(x, y) \rightarrow \#(z | A(z, x) \wedge A(z, y)[> | < | =]n)$$

Listing 11. OCL invariant to enforce Custom Existential Composition.

```
context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()->select( z |
z.aTarget->asSet()->contains(y) and z.aTarget->asSet()->contains(self))->size() [>|<|=]n)
```

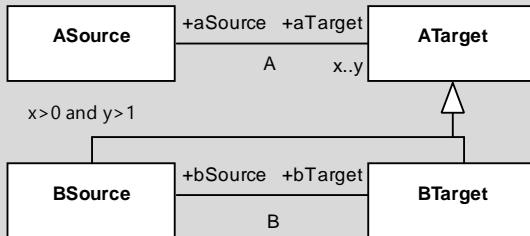
We summarize the description of the RelComp anti-pattern in Table 44.

Table 44. Characterization summary of the RelComp anti-pattern.

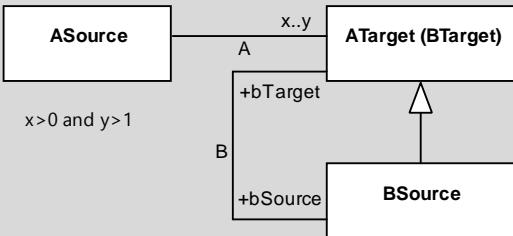
Name (Acronym)		Description
Relation Composition (RelComp)		<p>Consider two associations, no matter their stereotypes:</p> <ul style="list-style-type: none"> • A, that connects ASource and ATarGet; and • B, that connects BSource and BTarGet <p>For this anti-pattern to occur, one of the possible statements needs to be true:</p> <ul style="list-style-type: none"> • BSource equals or is a subtype of ATarGet and BTarGet equals or is a subtype of ATarGet. • BSource equals or is a subtype of ASource and BTarGet equals or is a subtype of ASource.
Type	Feature	Justification
Logical	Association	The instantiation of the two relations identified in this anti-pattern may restrict one another.
Pattern Roles		
Mult.	Name	Possible Types
1	A	All association stereotypes
1	ASource	All class stereotypes
1	ATarGet	All class stereotypes
1	B	All association stereotypes
1	BSource	All class stereotypes
1	BTarGet	All class stereotypes
Additional Constraints		
<ol style="list-style-type: none"> 1. A and B are different associations 2. The association A must have a minimum cardinality greater than 0 and a maximum greater than 1 in the association end connected to ATarGet. 3. One of the following sentences must evaluate to true: $(BSource = ATarGet \vee ancestorOf(ATarGet, BSource)) \wedge (BTarGet = ATarGet \vee ancestorOf(ATarGet, BTarGet))$ $(BSource = ASource \vee ancestorOf(ASource, BSource)) \wedge (BTarGet = ASource \vee ancestorOf(ASource, BTarGet))$ 		

Generic Example*

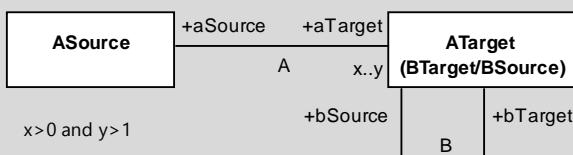
Variation 1



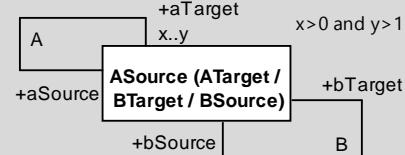
Variation 2



Variation 3



Variation 4



*Note: the presented variations are illustrative and do not intend to cover all possibilities

Refactoring Plans

1. **[OCL] Set Existential Composition:** add an OCL invariant to enforce that type B has an existential composition to type A:

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()-
>exists( z | z.aTarget->asSet()->contains(self) and z.aTarget-
>asSet()->contains(y) )
  
```

2. **[OCL] Set Right universal Composition:** add an OCL invariant to enforce that type B has a right universal composition to type A:

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()-
>forAll( z | z.aTarget->asSet()->contains(self) implies z.aTarget-
>asSet()->contains(y) )
  
```

3. **[OCL] Set Left Universal Composition:** add an OCL invariant to enforce that type B has a left universal composition to type A:

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()-
>forAll( z | z.aTarget->asSet()->contains(y) implies z.aTarget-
>asSet()->contains(self) )
  
```

4. **[OCL] Set Forbidden Composition:** add an OCL invariant to enforce that type B has a forbidden composition to type A:

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()-
>forAll( z | not(z.aTarget->asSet()->contains(y) and z.aTarget-
>asSet()->contains(self)) )
  
```

5. **[OCL] Set Custom Existential Composition:** add an OCL invariant to enforce that type B has a custom existential composition to type A:

```

context BSource
inv: self.bTarget->asSet()->forAll( y | ASource.allInstances()-
>select( z | z.aTarget->asSet()->contains(y) and z.aTarget-
>asSet()->contains(self) )->size() [>|<|=] n
  
```

Anti-Pattern Relations

Group by Feature (Association): AssCyc, BinOver, ImpAbs, RelSpec

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelSpec, RepRel

Causes: none

Caused by: none

5.15 RELATOR MEDIATING OVERLAPPING TYPES (RELOVER)

The Relator Mediating Overlapping Types (RelOver) is another purely logical anti-pattern. A relator connected to two or more overlapping types, through mediation associations, characterizes an occurrence. The concept overlapping types adopted here is the one defined for the BinOver anti-pattern. Informally, two or more types overlap if there is a possible instantiation of the model in which an individual instantiate all types simultaneously. For more details on how two types can overlap, please refer to BinOver's definition.

In addition, the sum of the mediations' upper bound cardinalities on the mediated end (opposite to the end connected to the relator) must be greater or equal to 2. This is required to reduce the number of "false alarms", since every relator instance must mediate at least two distinct individuals, as defined in OntoUML.

This modeling structure is prone to be overly permissive, since there is no restriction for an instance to act as multiples roles for the same relator. The possible commonly identified intended interpretations are that:

- the mediated types are actually **disjoint**, i.e., regardless of the relator, there is no individual that can even instantiate more than one of the mediated types;
- all mediated types are **exclusive**, i.e. objects can simultaneously instantiate more than one mediated type, but what they cannot do is play more than one role in the context of the same relator instance; and
- **partially exclusive** mediated types, a weaker version of the previous alternative, in which some roles can be simultaneously played, whilst other cannot.

Table 45 summarizes the characterization of the RelOver anti-pattern.

Table 45. Characterization summary of the RelOver anti-pattern.

Name (Acronym)		Description
Relator Mediating Overlapping Types (RelOver)		A relator connected, through mediations, to two or more types whose extension possibly overlap. The sum of the mediations' upper bound cardinalities of the mediated end must be greater than 2.
Type	Feature	Justification
Logical	Relator	This structure is usually too permissive. It is often the case that some of the mediated types should be disjoint or set as exclusive in the context of a single relator instance.
Pattern Roles		
Mult.	Name	Possible Types
1	Relator	«relator»
2..*	med-n	«mediation»
2..*	Over-n	All object types: «kind», «collective», «quantity», «subkind», «phase», «role», «roleMixin», «category» and «mixin»
Additional Constraints		
<ol style="list-style-type: none"> Let M be the set of identified mediations, mediatedEnd(m) the function that returns the association end opposed to relator of a mediation m, and upper(p) the function that return the upper bound cardinality of a property p, then: $\left(\sum_{m \in M} \text{upper}(\text{mediatedEnd}(m)) \right) > 2$ <ol style="list-style-type: none"> Let O be the set of types mediated by Relator, then: $\exists x, y \in O \mid \text{overlap}(x, y)$ 		
Generic Example*		
<p>*Note: the presented variations are illustrative and do not intend to cover all possibilities</p>		

Refactoring Plans

1. **[OCL] Exclusiveness***: choose this option to forbid the same individual to play multiple roles w.r.t the same relator instance. Create an OCL invariant according to the following template:

```
context Relator
inv: self.over1.oc1AsType(Supertype) ->asSet() ->excludesAll(
    self.over2.oc1AsType(Agent) ->asSet() and
    self.over1.oc1AsType(Supertype) ->asSet() ->excludesAll(
        self.over3.oc1AsType(Agent) ->asSet() and
        self.over2.oc1AsType(Supertype) ->asSet() ->excludesAll(
            self.over3.oc1AsType(Agent) ->asSet() ))
```

2. **[OCL] Partially exclusiveness**: choose this option to forbid a subset of mediated types as exclusive.
3. **[Mod/New] Disjoint mediated**: Enforce types to be disjoint through the creation or alteration of a disjoint generalization set.

**Note: to make all types exclusive, every binary combination should be explicitly ruled out*

Anti-Pattern Relations

Group by Feature (Relator): DepPhase, FreeRole, MultDep, RelRig, RepRel

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelComp, RelRig, RelSpec, RepRel

Causes: UndefFormal (3), AssCyc (3)

Caused by: none

Figure 39 depicts a fragment of UFO-S, a commitment-based core reference ontology about services (NARDI et al., 2013). The fragment provides a partial description of the concepts of “Service Offering” and “Service Agreement”. A provider makes an offering, which describes the terms in which she will provide the service. The agreement formalizes that a customer and a provider already negotiated the terms of for hiring a service.

The authors exemplify UFO-S using the car rental domain. A car rental company acts as a “Service Provider”, when offering to rent cars. Their “Target Customer Community” contains, as members, all adults. When “Luke”, for example, decides to rent a car from a particular company and signs the rental agreement, he acts as the “Service Customer” and the company as the “Hired Service Provider”. The rental agreement is the “Service Agreement”, which specifies the conditions in which the service was hired (price, duration, insurance and so on).

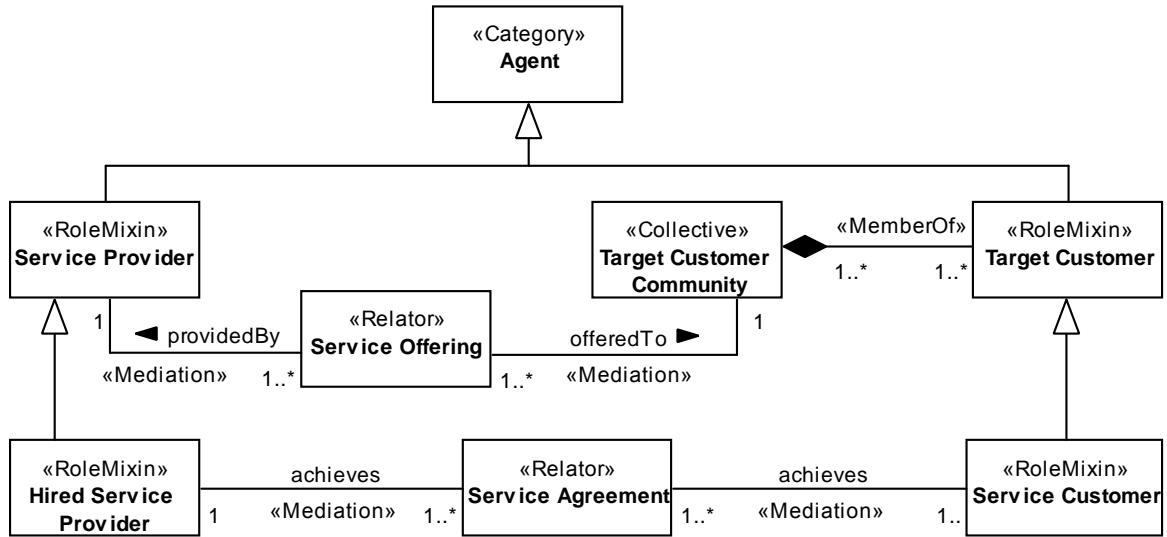


Figure 39. RelOver occurrence encountered in the UFO-S ontology.

The relator “*Service Agreement*” characterizes the RelOver occurrence, because it is the truth-maker of a material relation involving the overlapping types “*Hired Service Provider*” and “*Service Customer*”. Note that, although the upper multiplicity on the provider side is one, the upper bound multiplicity on the customer side is unlimited.

The analysis of any RelOver occurrence starts by verifying the modeler desires disjointness for the mediated types. In our example, the inquiry is whether it is possible for a provider to be a customer also (e.g., a car rental company hiring the services of an accounting company). We assume here that the answer is yes, i.e. no disjointness constraint is required.

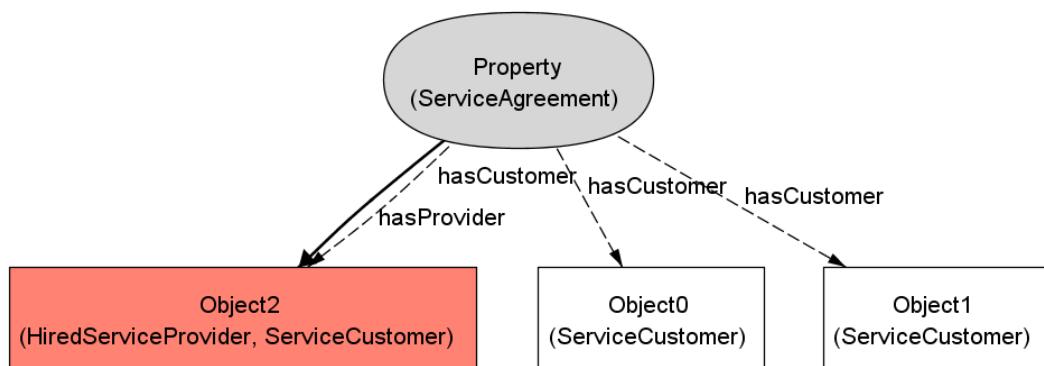


Figure 40. Overlapping mediated types without exclusiveness constraint.

Keeping the mediated types as overlapping allows situations like the one depicted in Figure 40 – a world in which one individual is both the provider and the customer in the

context of the same agreement. If the authors decide to forbid such instantiations, they should enrich their ontology with the OCL invariant in Listing 12.

Listing 12. OCL invariant to enforce exclusive mediated types

```
context 'Service Agreement'
inv exclusiveTypes: self.provider.oclAsType(Agent) ->asSet() ->excludesAll(
self.customer.oclAsType(Agent) ->asSet())
```

To complete our example, Figure 41 shows a possible model instantiation still allowed after adding the exclusiveness constraint. “Object1” and “Object2” play both the provider and the customer roles, but now in the context of different agreements. “Object0”, conversely, is just a customer in both agreements.

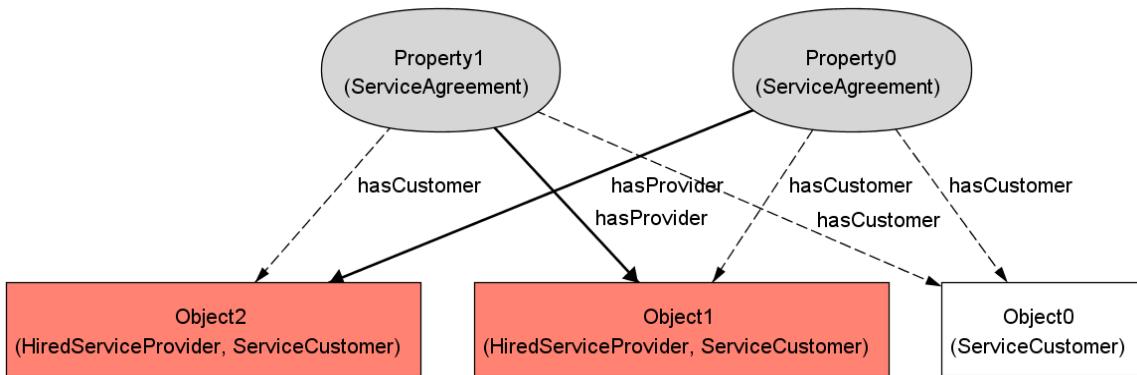


Figure 41. Simultaneous role instantiation with exclusive relators

5.16 RELATOR MEDIATING RIGID TYPES (RELRIG)

The Relator Mediating Rigid Types (RelRig) ant-pattern occurs whenever a model contains a relator connected to at least one rigid type (stereotyped as «kind», «quantity», «collective», «subkind» or «category») through associations stereotyped as «mediation».

When a type is connected to a mediation association, it means that it is externally dependent, i.e. for an individual to instantiate it, it must be related to another type. Usually, mediations define roles and roleMixins – anti-rigid types. When modelers formalize mediations between a relator and a rigid object type (stereotyped as kind, collective, quantity, subkind and category) some situations should be further investigated.

The analysis begins with the verification of the mediated type rigidity, which can be decided by answer the following question: “Can an individual that was not created as <Type> to become one, or an individual that is already an instance of <Type> cease to be it and still exists?”. If the answer is yes the type is anti-rigid, otherwise it is rigid. Now, if the mediated type turns out to be anti-rigid, the solution is simply to change its stereotype to role, if it was originally a sortal, or as RoleMixin, if it was originally a non-sortal. Note that, if the mediated type was an identity provider, it will require a new one.

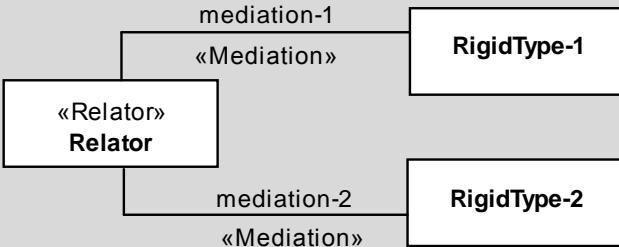
In the cases that the mediated types is indeed rigid, one must verify if the relational dependency captured by the mediation is indeed mandatory. If it is not, the creation of a role or roleMixin is in order. Please refer to the definition of the MultDep anti-pattern for more details regarding optional and mandatory relational dependencies.

In the cases where the mediated types are rigid and the dependencies mandatory, what comes to analysis is the direction of the existential dependency. As discussed in (REF), existential dependency is a particular type of dependency for which an individual (the dependent) depends on another individual (the dependee) to exist. For example, a person is existentially dependent on their brain and a car is existentially dependent on its chassis. Furthermore, relators are always existentially dependent of the individuals they relate. A marriage, for example, always involve the same people, otherwise is a different marriage.

With all that in mind, the modeler should verify if the direction of the dependency. If the relator depends on the mediated but not the opposite, the *Re/Rig* occurrence is a false alarm and the modeler can ignore it. If the dependency goes only from the mediated to the relator, the stereotype of the association and the stereotype of the rigid type are wrong. In this case, the modeler should transform the mediation into a characterization and the mediated type into a mode (for more details about mode and characterizations please refer back to Chapter 2). Lastly, if the dependency goes both ways, the modeler needs to set the meta-property *isReadOnly* in the mediated end of the association to true.

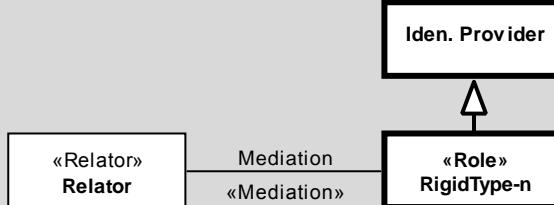
We make a summary of the RelRig anti-pattern on Table 46.

Table 46. Characterization summary of the *RelRig* anti-pattern.

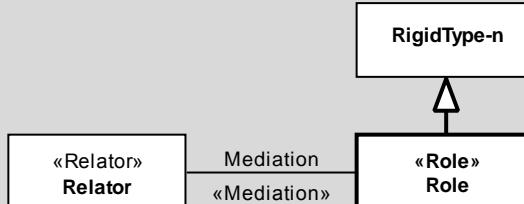
Name (Acronym)		Description
Relator Mediating Rigid Types (RelRig)		A relator connected to one or more rigid types through mediations.
Type	Feature	Justification
Logical; Scope	Relator	When a type is connected to a mediation association, it means that it is externally dependent, i.e. for an individual to instantiate it, it must be related to another type. Usually, mediations define roles and roleMixins – anti-rigid types.
Pattern Roles		
Mult.	Name	Possible Types
1	Relator	«relator»
1..*	mediation-n	«mediation»
1..*	RigidType-n	«kind», «quantity», «collective», «subkind» and «category»
Additional Constraints		
<ol style="list-style-type: none"> Let $\text{relator}(m)$ and $\text{mediated}(m)$ be the functions that return, respectively, the relator and the mediated types connected to a mediation. Also, let M be the set of mediation-n and R the set of RigidType-n, then: $\forall m \in M, \text{relator}(m) = \text{Relator} \wedge \text{mediated}(m) \in R$ Let $\text{mediatedEnd}(m)$ be the function that returns the association end connected to the mediated type of a given mediation m, $\text{isReadOnly}(p)$ the function that return the value of the isReadOnly meta-property of an association end p and M the set of the identified mediations, then: $\forall m \in M, \text{isReadOnly}(\text{mediatedEnd}(m)) = \text{true}$ 		
Generic Example		
 <pre> classDiagram class Relator { <<Relator>> } class RigidType1 { <<RigidType-1>> } class RigidType2 { <<RigidType-2>> } Relator "1" -- "1..*" mediation-1 : "«Mediation»" Relator "1" -- "1..*" mediation-2 : "«Mediation»" mediation-1 --> RigidType1 mediation-2 --> RigidType2 </pre>		

Refactoring Plans

1. **[Mod/New] Set as role:** choose this plan when a RigidType-n should be anti-rigid. If previously stereotype with a sortal stereotype, change it to role, if non-sortal, change to roleMixin. (If RigidType-n was stereotyped as kind, collective or quantity, a new identity provider should be created for it using the same stereotype).



2. **[New/Mod] Add role subtype:** choose this action if the mediation-n is optional for RigidType-n. Create a role (for sortals) or a roleMixin (for non-sortals) that specializes RigidType-n and move mediation-n to it.



3. **[Mod] Set as mode:** choose this plan when RigidType-n is in fact an unstructured property of Relator-n. This is only true if the existential dependency specified in the mediation is reversed (RigidType-n should depend on Relator and not the other way around)
4. **[Mod] Set bidirectional existential dependency:** choose this action if the event that creates the relator is the same one that creates RigidType-n and also this relation established in the individuals creation may never change

Anti-Pattern Relations

Group by Feature (Relator): DepPhase, FreeRole, MultDep, RelOver, RepRel

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RepRel

Group by Type (Scope): DepPhase, FreeRole, GS Rig, ImpAbs, HomoFunc, MixIden, MixRig, MultDep, UndefPhase

Causes: UndefFormal (3), AssCyc (3)

Caused by: none

Figure 42 presents a model fragment that characterizes a RelRig occurrence. The diagram was adapted from the ECG ontology (GONÇALVES et al., 2007) and describes that a heart's contraction mechanism. A regular heart is composed by ventricles and atriums, cavities from which the blood is pumped through. The pumping action is caused by contractions, consequences of electrical impulses generated by the heart's pacemakers cells.

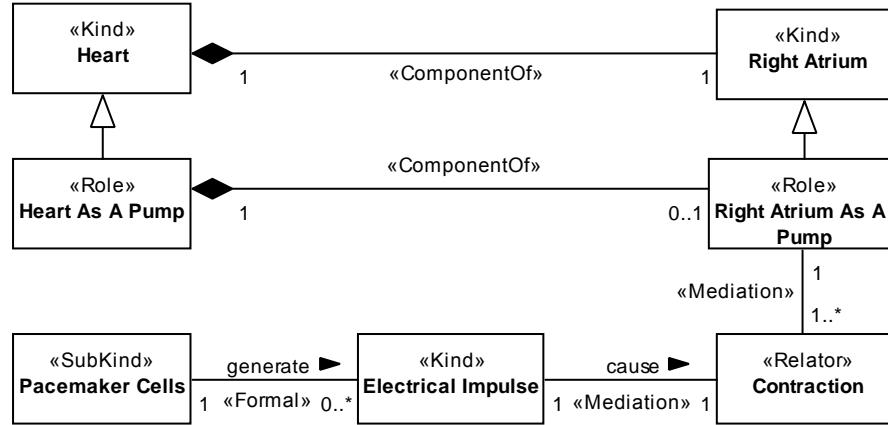


Figure 42. Simplified fragment of the ECG ontology characterizing a RelRig occurrence.

The *RelRig* occurrence is characterized by the relator *Contraction* connected to the kind *Electrical Impulse*. To verify whether or not it characterizes a mistake, we go through the proposed analysis process. An impulse is always an impulse, so the rigidity is correct. Furthermore, the generation of the impulse implies in an immediate contraction of the ventricle, so the relation is indeed mandatory. Now, the direction of the existential dependency remains for analysis. A contraction is always a consequence of the same electrical impulse, but an impulse only causes the same contraction whilst it exists. Therefore, we encounter a bidirectional existential dependency. To refactor the model to obtain that, one would only need to set the mediated end of mediation “cause” readOnly property to true.

5.17 RELATION SPECIALIZATION (RELSPEC)

The Relation Specialization Anti-Pattern (RelSpec) consists of two relations A and B that connect types ASOURCE and ATARGET, and BSOURCE and BTARGET, respectively, such that one of the following conditions holds:

- ASOURCE is equal to or a subtype of BSOURCE, and ATARGET is equal to or a subtype of BTARGET;
- ASOURCE is equal to or a subtype of BTARGET, and ATARGET is equal to or a subtype of BSOURCE

Our empirical studies showed the structures identified by this anti-pattern are likely to require additional constraints, which restricts the instantiation of associations A and B.

Four different types of restriction are recurrent: subsetting, redefinition, association disjointness and specialization.

It is true that association B **subsets** association A if being related through B implies being related through A but not the other way around. To exemplify, consider the relations of being a father and of being an ancestor, which hold between people. It is true that “father of” subsets “ancestor of” because every father is an ancestor, but not every ancestor is a father.

If the *Re/Spec* occurrence requires a subsetting constraint, the modeler should add one of the A’s association ends, the subsetted association, to the subsetted properties list of the respective association end of B. The formal semantics of a subsetted property is described in (COSTAL; GÓMEZ; GUIZZARDI, 2011) as in Listing 13.

Listing 13. Subsetting constraint written in OCL.

```
context BSource
inv subset : self.oclAsType(ASource).aTarget->includesAll(
    self.bTarget.oclAsType(ATarget))
```

Association B **redefines** association A if, and only if, whenever an individual instantiates BSource, the individuals it is related through B are the same individuals it is related through A. Note that, like subsetting, in redefinitions, being related through B also implies being related through A. The difference is that there cannot be individuals related through the “parent” association but not through the “child” one.

Analogous to subsetting, OntoUML’s meta-model specifies a list of redefined properties for each association end. The formal semantics of a redefined property is also defined in (COSTAL; GÓMEZ; GUIZZARDI, 2011) as in Listing 14.

Listing 14. Redefinition constraint written in OCL.

```
context BSource
inv redefinition : self.oclAsType(ASource).aTarget =
    self.bTarget.oclAsType(ATarget)
```

We make a caveat for adopting the redefinition constraint when A and B relate the same types. In these cases, the extension of the associations will always be same and they will turn out to be redundant relations, increasing the model’s complexity without providing new knowledge. From there, the modeler can take two alternative paths:

delete one of the associations and forget about the redefinition constraint; or specialize at least one of B's end and keep the redefinition constraint.

The third type of constraint that identified between associations A and B is **disjointness**. In this case, B is disjoint from A if being related through B implies not being related through A. To exemplify, consider a queue and the relations of predecessor and successor, which hold between individuals in the queue. If an individual is the predecessor of another, it implies that it is not its successor.

The OntoUML meta-model does not consider the possibility of disjoint relations. For that reason, to enforce a constraint of such nature, the OCL invariant presented in Listing 15 should enrich the model.

Listing 15. Disjointness constraint written in OCL.

```
context BSource
inv disjoint : self.oclAsType(ASource).aTarget->excludesAll(
    self.bTarget.oclAsType(ATarget))
```

The last refactoring plan for the RelSpec anti-pattern is to make B a **specialization** of A. As showed in (COSTAL; GÓMEZ; GUIZZARDI, 2011), specializing and subsetting have the same formal semantics, the inclusion constraint of B in A. However, specialization represents an intentional relation between types, i.e., every property a parent relation has, its child relation will inherit. Furthermore, the event that establishes both relations is also the same. To specify this constraint, one just needs to create a generalization between the relations (from B to A).

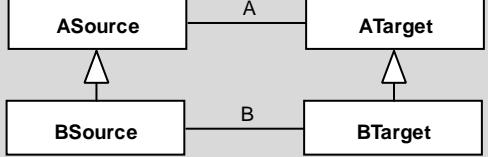
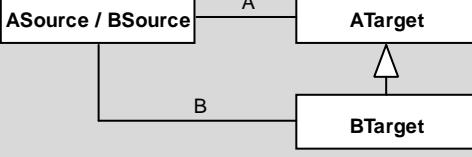
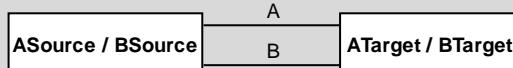
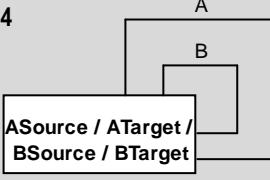
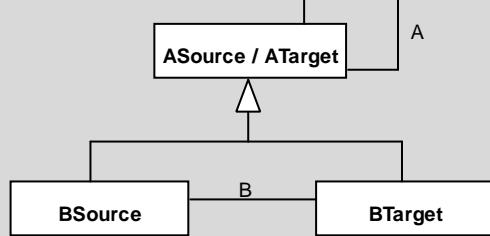
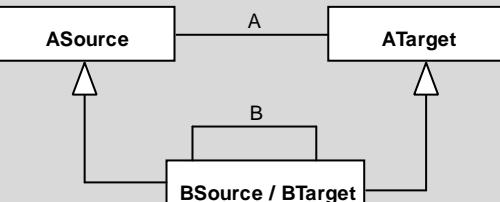
We present the overview of the RelSpec anti-pattern in Table 47.

Table 47. Characterization summary of the RelSpec anti-pattern.

Name (Acronym)		Description
Relation Specialization (RelSpec)		<p>Two associations A, connecting ASource to ATarGet, and B, connecting BSource to BTarGet, such that:</p> <ul style="list-style-type: none"> • ASource is equal or a subtype of BSource and ATarGet is equal or a subtype of BTarGet; or • ASource is equal or a subtype of BTarGet and ATarGet is equal or a subtype of BSource
Type	Feature	Justification
Logical	Association	The identified structure suggests the existence of a specialization between the relations or the need for including a subsetting, redefinition or disjoint constraint.

Pattern Roles		
Mult.	Name	Possible Types
1	A	All association stereotypes
1	ASource	All class stereotypes
1	ATarget	All class stereotypes
1	B	All association stereotypes
1	BSource	All class stereotypes
1	BTarget	All class stereotypes

Additional Constraints		
1.	A and B are different associations	
2.	One of the following sentences must evaluate to true: $(ASource = BSource \vee ancestorOf(ASource, BSource))$ $\wedge (ATarget = BTTarget \vee ancestorOf(ATarget, BTTarget))$ $(ASource = BTTarget \vee ancestorOf(ASource, BTTarget))$ $\wedge (ATarget = BSource \vee ancestorOf(ATarget, BSource))$	

Generic Example*		
Variation 1		Variation 2
		
Variation 3		Variation 4
		
Variation 5		Variation 6
		

**Note: the presented variations are illustrative and do not intend to cover all possibilities*

Refactoring Plans

1. **[Mod] Subset:** this action should be taken if being connected through relation B implies being connected through relation A but not the other way around. The fix consists in adding one of A's association ends to the subsetted properties of B's respective association end. Alternatively, the following OCL can be included in the model*:

```
context BSource
inv subset : self.oclAsType(ASource).aTarget->includesAll(
    self.bTarget.oclAsType(ATarget)
```

2. **[Mod] Redefine:** this action should be taken if being related through B implies not only being related through A but requiring that all related elements through A are related through B. The fix consists in adding one of A's association ends at the redefined properties set of B's respective association end. Alternatively, the following OCL can be included in the model*:

```
context BSource
inv subset : self.oclAsType(ASource).aTarget=
    self.bTarget.oclAsType(ATarget)
```

This solution is strongly discouraged if associations A and B related the same types.

3. **[Mod/New] Disjoint:** this action should be taken if being related through B implies not being related through A. Differently from the first two, this constraint can only be enforce through OCL invariants:

```
context BSource
inv subset : self.oclAsType(ASource).aTarget->excludesAll(
    self.bTarget.oclAsType(ATarget)
```

4. **[New] Specialize:** the logical implication of this solution is the same as enforcing subsetting. Nonetheless, it should only be selected if association B is a particular type of A and not only if the logical constraint is required.

*Assuming that the occurrence is the structural variation number 1

Anti-Pattern Relations

Group by Feature (Association): AssCyc, BinOver, ImpAbs, RelComp

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelRig, RepRel

Causes: none

Caused by: MultDep

Figure 43 depicts a RelSpec occurrence identified in the OntoBio ontology (ALBUQUERQUE, 2011). The diagram presents the different relations between the concepts “Environment” and “Spatial Location”. An “Environment” provides the biological characteristics, like vegetation, soil composition and climate, for a region delimited by geographical coordinates (defined by latitude, longitude and altitude). If a single coordinate defines a location, the authors named it a “Geographic Point”. Furthermore, a “Micro Environment” characterizes it. Analogously, multiple coordinates (a region) define a “Geographic Space”, whom a “Macro Environment” characterizes.

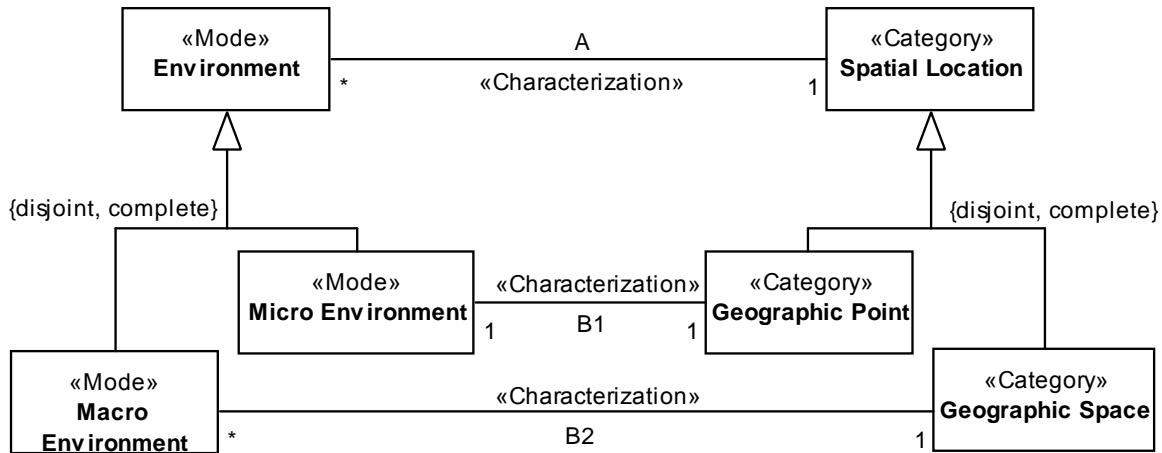


Figure 43. RelSpec occurrence identified in the OntoBio ontology.

In the diagram, we find two RelSpec occurrences: one composed of the characterizations A and B1 and another by characterizations A and B2. Now, we present possible instantiation allowed by models restricted using each of the constraints. For simplicity reasons, we only demonstrate scenarios using characterizations A and B2.

Figure 44 presents a valid world if B2 redefines A. Note that, “Object1”, an instance of “Geographic Space”, relate to the same individuals through B2 and A. “Object0”, conversely, is not an instance of “Geographic Space”, only of “Spatial Location”, and thus the same restriction does not apply to it.

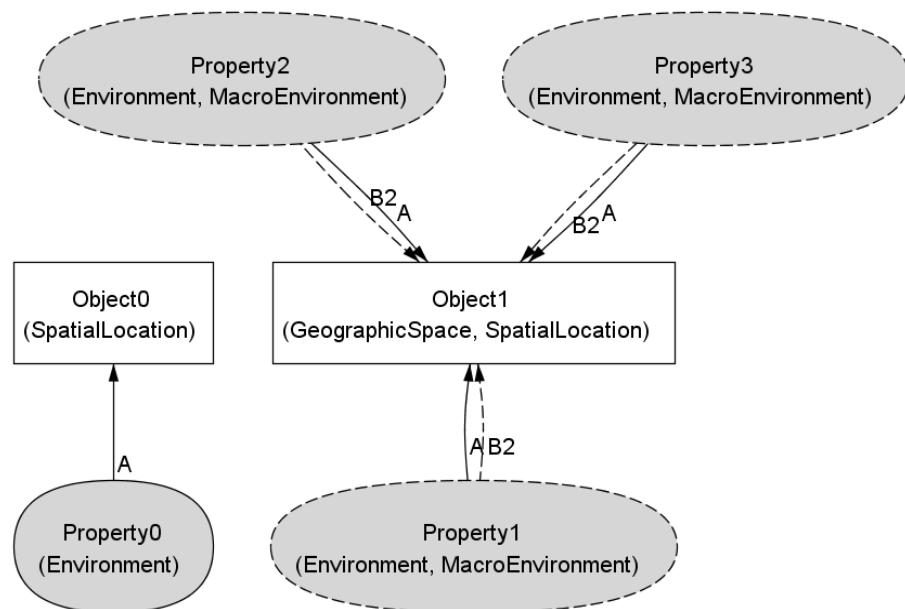


Figure 44. Characterization of the redefinition constraint.

The possible instantiation depicted in Figure 45 exemplifies the implications of the subsetting constraint. As desired, the inclusion constraint of B2 in A is there: for all macro environments that the individual named “Object” is connected through B2, it is connected through A. However, “Object” is connected to “Property0” only though A, since the implication is not applied in both ways.

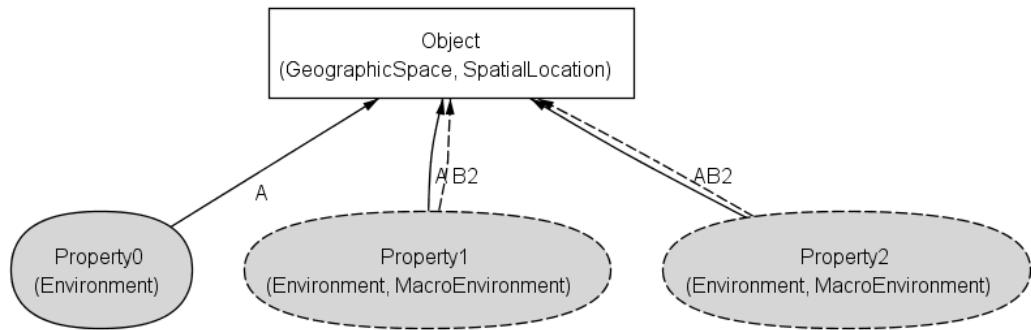


Figure 45. Characterization of the subsetting constraint.

Lastly, we present an exemplification of enforcing the disjoint constraint in Figure 46. Differently from the other scenarios, whenever a “Geographic Space” is connected to a “Macro Environment” through B2 it is not connected through A. “Object1”, for example, is characterized by the environments “Property0” and “Property1” through A and by “Property2” through B2.

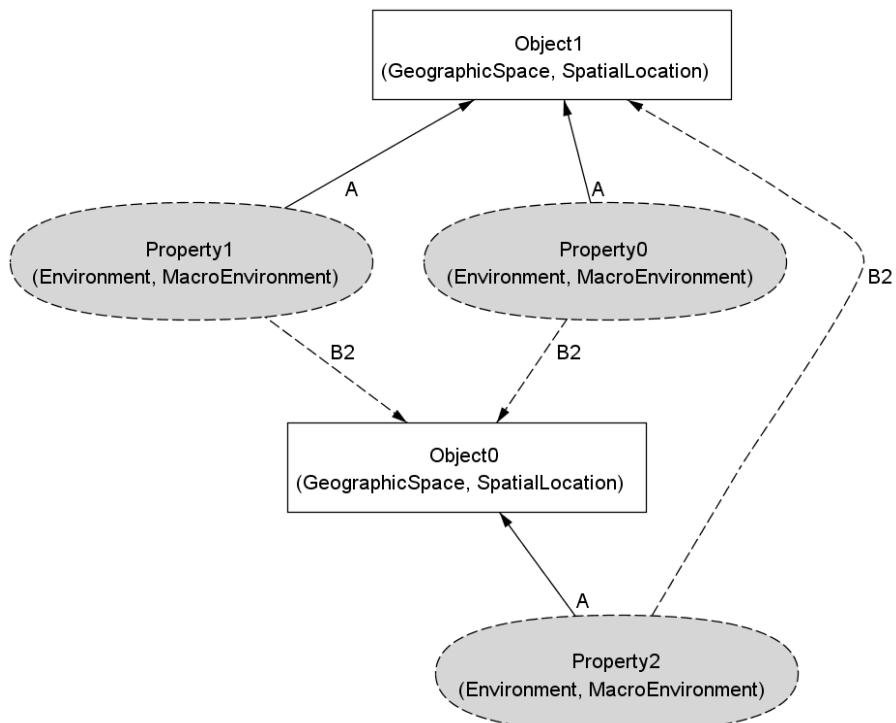


Figure 46. Characterization of the disjointness constraint.

The adopted solution for this RelSpec occurrence in the OntoBio ontology was to enforce the redefinition constraint on both relations B1 and B2.

5.18 REPEATABLE RELATOR INSTANCES (REPREL)

The Repeatable Relator Instances Anti-pattern (RepRel) is an application of an Object-role Modeling (ORM) construct, named internal uniqueness constraint, in OntoUML. ORM is a fact-based modeling approach for expressing information at conceptual level (HALPIN; MORGAN, 2008). As its name suggest, objects and roles are the language's core. Moreover, predicates define roles (ORM allows unary, binary, ternary predicates and so one). For example, to represent that a person can be a smoker, one would represent an entity, Person, with a unary predicate, smokes. To represent that a Person owns a Car, one would model two object types, Person and Car, and a binary predicate between them, which defines the roles of "being owner" and of "being owned".

In ORM, a modeler can apply the internal uniqueness constraint to predicates to limit the number of identical combinations in the predicate. For example, if applied to the binary predicate "owns" that we previously defined, it would forbid a person to own a car more than once. Extensions for the "owns" predicate like {(John, Car1), (Joseph, Car2), (Luke, Car3)} would be accepted, but {(John, Car1), (Joseph, Car2), (John, Car1)}, would not.

OntoUML, like UML, is not fact-oriented. Nonetheless, that does not mean that the expressivity of the uniqueness constraint would not be beneficial for it. Instead of predicates that define roles for entities, OntoUML provides use relators for that purpose. Just like predicates, relators connect two or more types. A priori, the number of relator instance copies (instances that mediate the same individuals) is not limited. The goal of the *RepRel* anti-pattern is to drive modelers into specifying restrictions like the Uniqueness Constraint.

Throughout our investigations, however, we identified a finer grained distinction required to specify the limit of coexistent relator copies. That distinction comes from

two very different intended semantics for the relator, what we name current and historical relators.

On one hand, a **current relator** is one whose instantiation corresponds to the existence of the individual, i.e. if a relator is instantiated in a world it is because it exists in that world. On the other hand, **historical relators** are the ones that intended to capture a registry of the existence of a relator. Furthermore, since the existence does not correspond to the instantiation, modeler might desire to reify the relator's existence. One can achieve that using “active” and “inactive” phases or through time stamps to identify the point in time in which the relator is created and destructed. Lastly, modelers might desire to enforce “eternal” semantics to historical relators, i.e., relators that, after created, never disappear.

To exemplify the distinction between historical and current relators, consider the relator marriage (for simplicity, let us assume monogamous marriages in this example) that defines the husband and wife roles. By law, it is only possible for a man to be married to exactly one wife at a time and vice-versa. Nonetheless, throughout one's life, one can marry again, if properly divorced. If this domain is modelled using current semantics for the relator, the cardinality would be exactly one on the relator end (and that would still allow many marriages throughout time). Conversely, if the modeler assumes an historical view, the multiplicity would be one or more and the relator existence.

To analyze an occurrence of the *RepRel* anti-pattern, a modeler must decide which of the aforementioned semantics she intends for the relator. If it is current, adding the following OCL invariant in Listing 16 will restrict the number of repeated instances.

Listing 16. OCL version of the Uniqueness Constraint for “current” relators.

```
context Relator
inv: Relator.allInstances()->select( r | r>>self and r.type1=self.type1 and
r.type2=self.type2)->size()=<n-1>
```

Enforcing uniqueness constraints on historical relators is more complex. We propose the existence reification through the creation of attributes to act like time stamps, named “start”, to identify the creation time, and “end”, to identify destruction. Since OntoUML does not specify a data type library, one would to create their own Time data type. Additionally, the modeler should add the OCL code provided in Listing 17.

Listing 17. Enforcing Uniqueness Constraint for “historical” relators using OCL.

```

context Relator
inv: Relator.allInstances()->select( r | r><self and r.type1=self.type1 and
r.type2=self.type2 and self.concurrent(r))->size()=<n-1>

context Relator::concurrent(r:Relator) :Boolean
body: self.start=r.start or
        (self.start<r.start and r.start<self.end) or
        (r.start<self.start and self.start<r.end)

```

We provide a complete summary of the RepRel anti-pattern in Table 48.

Table 48. Characterization summary of the RepRel anti-pattern.

Name (Acronym)		Description
Repeatable Relator Instances (RepRel)		A «relator» connected to two or more «mediation» associations, whose upper bound cardinalities at the relator end are greater than one.
Type	Feature	Justification
Logical		
Relator		Inspired in ORM’s uniqueness constraint (HALPIN; MORGAN, 2008), this anti-pattern aids the modeler in specifying the number of different relators instances that can mediated the exact same set of individuals.
Pattern Roles		
Mult.	Name	Possible Types
1	Relator	«relator»
2..*	med-n	«mediation»
2..*	Type-n	All object stereotypes: «kind», «quantity», «collective», «subkind», «role», «phase», «roleMixin», «mixin» and «category»
Additional Constraints		
<ol style="list-style-type: none"> Let M be the set of the mediations that characterize RepRel, relatorEnd(m) the function that return the association end whose type is the relator of a mediation m, and upper(p) the function that return the upper bound cardinality of a property p, then: $\forall m \in M, \text{upper}(\text{relatorEnd}(m)) > 1$ <ol style="list-style-type: none"> Let M be the set of the mediations that characterize RepRel, relator (m) the function that returns the relator connected to a mediation m, then: $\forall m \in M, \text{relator}(m) = \text{Relator} \vee \text{isAncestor}(\text{relator}(m), \text{Relator})$ $\exists m \in M, \text{relator}(m) = \text{Relator}$		
Refactoring Plans		
<ol style="list-style-type: none"> [Mod] Fix upper cardinality: this plan is individually to the mediations. It consists in changing the maximum cardinality on the relator to a usually lower value. [OCL] Define uniqueness constraint (Current Relator): this plan is applied to a combination of the mediations. Although it can be applied more than once, for different 		

combinations, it cannot be applied simultaneously with the historical relator plan. This should be taken if there is a limit of the number of coexistent relator instances that mediated the same combination of the mediated types. The following OCL invariant should be created (where $<n>$ is the limit of “cloned” relators):

```
context Relator
inv: Relator.allInstances()->select( r | r<>self and
r.type1=self.type1 and r.type2=self.type2)->size()=<n-1>
```

3. **[OCL] Define uniqueness constraint (Historical Relator):** this plan applies to a combination of the mediations and, although it can be applied more than once for different combinations, it cannot be applied simultaneously with the current relator plan.

```
context Relator
inv: Relator.allInstances()->select( r | r<>self and
r.type1=self.type1 and r.type2=self.type2 and concurrent(self,r))->size()=<n-1>
context Relator::concurrent(r:Relator) :Boolean
body: self.start=r.start or
(self.start<r.start and r.start<self.end) or
(r.start<self.start and self.start<r.end)
```

Anti-Pattern Relations

Group by Feature (Relator): DepPhase, FreeRole, MultDep, RelOver, RelRig

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, WholeOver, RelOver, RelComp, RelSpec, RelRig

Causes: none

Caused by: none

Figure 47 depicts a RepRel occurrence identified in the Configuration Management Ontology (CMTO) (CALHAU; FALBO, 2012). The fragment focuses on the relator “Change Request”, which captures a registry of the action made by a “Requester”, when soliciting changes in one or more versions of a configuration item.

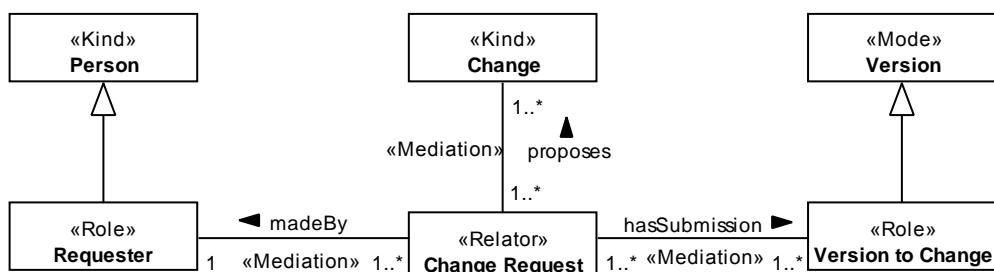


Figure 47. RepRel occurrence identified in the CMTO ontology.

The authors explicit use the word “register” when describing the relator class “Change Request”. Furthermore, they make an observation that there should be a quality to identify the request’s time of creation, even though they did not explicitly represented it in the model. Considering these two facts along with the purpose of the ontology, provide support for the integration of configuration management system, we

reasonably conclude that the authors intend an historical semantics for the relator “*Change Request*”.

Continuing the analysis of this fragment, we use the simulation to generate examples and encounter the possibility depicted in Figure 48. As it is noticeable, the ontology allows the same requester to make more than one request regarding the same change and the same version. If that is not desirable, one only needs to enrich the model with the OCL invariant described in Listing 18.

Listing 18. OCL constraint to limit repeated relators in the CMTO ontology.

```
context _'Change Request'
inv:_'Change Request'.allInstances()->select( r | r<>self and
r.requester=self.requester and r.version=self.version and
r.change=self.change and self.concurrent(r))->size ()=0
```

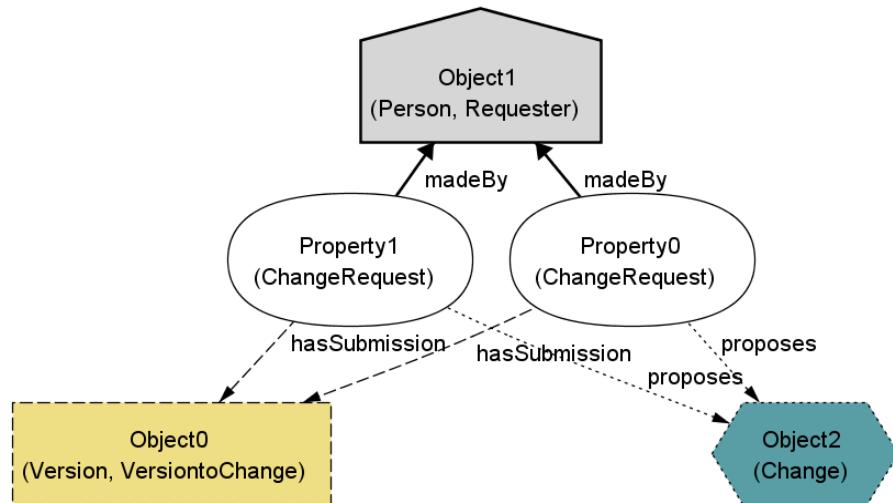


Figure 48. Possible world generated for the diagram in Listing 18 without adding the uniqueness constraint regarding the relator *Change Request*.

5.19 UNDEFINED FORMAL ASSOCIATION (UNDEFFORMAL)

UFO classifies relations in two main groups: material (or external) and formal (or internal). Briefly, the difference between them is that in order for a material relation to hold between two individuals, it requires an external entity: named its truth-maker. Formal relations, conversely, do not require such entity.

The meaning of the generic concept of formal relation is not the same of the one that motivated the creation of the «formal» stereotype. In fact, in this broader sense, all part-whole relations, mediations and characterizations are formal. OntoUML, however, defines the «formal» stereotype to formalize a particular subset of formal relations, named Domain Comparative Formal Relation (DCFR).

The DCFR captures relations reducible to the comparison of values from qualities (datatypes) that characterize the related types. An example of such relation is “heavier than”, which holds between two people and that can be derived from the comparison of their weights.

We argue that the DCFR’s definition requires the related types to own or inherit at least one quality, from which modelers can derive the relation. If that is not the case, one of the following affirmatives must be true: the ontology is missing the qualities required to derive the relation; or the formal relation is not a DCFR. The Undefined Formal Association (UndefFormal) proposes to investigate which of these possibilities is true.

Another motivation to investigate formal relations comes from our empirical experience in analyzing OntoUML models. Our hypothesis is that modelers use the formal stereotype as an “escape route” when they do not know which stereotype to use. In fact, since OntoUML does not restrict in any way the use of formal relations, a modeler randomly using a formal relation will always obtain syntactically valid models.

As expected, the UndefFormal’s refactoring plans are properly specifying the formal relation as a DCFR or changing its stereotype. If one decides to specify properly the DCFR, one needs to:

- specify the data types to which it will be derived from;
- set the relation as derived; and
- specify the OCL derivation rule;

If the chosen alternative is the stereotype change, the plan’s actions will heavily depend on the related types’ stereotypes. For example, if a formal relation relates a mode and a kind, the only other possible association stereotype is the characterization. If the modeler reaches the conclusion that the relation is formal, but not a DCFR, a

mediation, a characterization or any of the part-wholes, we conclude that it is a type of formal relation (in the general sense) that is not contemplated in the OntoUML syntax.

We consolidate the description of the UndefFormal anti-pattern in Table 49.

Table 49. Characterization summary of the UndefFormal anti-pattern.

Name (Acronym)		Description
Undefined Formal Association (UndefFormal)		A «formal» association defined between types that do not own or inherit quality properties, i.e., attributes or associations whose types are data types.
Type	Feature	Justification
Classification	Formal	Although OntoUML imposes no syntactical constraints on formal relations, it does not mean that modelers can use them at will, what is a very common practice.
Pattern Roles		
Mult.	Name	Possible Types
1	formal	«formal»
1	Source	All class stereotypes
1	Target	All class stereotypes
Additional Constraints		
<ol style="list-style-type: none"> Let qualities(c) be the function that return all qualities defined for a class c (through attributes or relations) and ancestor(c) be the function that return all direct and indirect super types of a class c, then: $\#qualities(Source) = 0 \wedge \forall x \in ancestor(Source), \#qualities(x) = 0 \wedge \#qualities(Target) = 0 \wedge \forall x \in ancestor(Target), \#qualities(x) = 0$		
Generic Example		
<pre> graph LR Target[Target] ---> Source[Source] style Target fill:#fff,stroke:#000 style Source fill:#fff,stroke:#000 style Source fill:#fff,stroke:#000 linkStyle 0 stroke:#000,stroke-width:1px linkLabel1["formal"] linkLabel2["«Formal»"] </pre>		
Refactoring Plans		
<ol style="list-style-type: none"> [New/Mod/OCL] Set as DCFR: choose this plan if the formal relation really is a DCFR. The fix consists in specifying the data types to which the relation will be derived from, set the relation as derived, and specify the OCL derivation rule. [Mod] Change stereotype: this alternative should be taken if one reaches the conclusion that the relation is better qualified by another stereotype. It consists only in changing the stereotype of the relation. 		

Anti-Pattern Relations

Group by Feature (Formal): none

Group by Type (Classification): DepPhase, GSRig, HetColl, HomoFunc, MixIden, MixRig, RelRig, UndefPhase

Causes: none

Caused by: none

Extracted from the Open proVenance Ontology (OvO) (CRUZ; CAMPOS; MATTOSO, 2012), the diagram depicted in Figure 49 exemplifies UndefFormal. The authors propose OvO as a reference model for the provenance domain, aiming to aid researchers understand and explore provenance meta-data.

The particular fragment depicted in the pictures describes the two roles researchers can play in the context of an experiment: being responsible for its execution and controlling it. The model also formalizes that a researcher, if acting as a coordinator, can conceive projects, which in turn have experiments as parts.

The diagram actually presents three occurrences of the UndefFormal anti-pattern. Each formal relation identifies a single occurrence: “executed by”, between “*Experiment*” and “*Researcher*”; “controlled by”, between “*Experiment*” and “*Coordinator*”; and “conceived by”, between “*Project*” and “*Coordinator*”.

We begin the analysis with the association “executed by”. Clearly, it cannot be reduced to a comparison of qualities of Researcher and Experiment, but is it formal in the general sense? If we assume that the intended meaning of the relation is to register who performed experiments in the past (but not who is currently performing one), we conclude that “executed by” is in fact a material relation derived from the “*Experiment Execution*” relator. Furthermore, the representation of the execution relator will solve an ambiguity issue on the relation, because it will specify whether one can perform many executions, but each for experiment, or if one can execute many experiments simultaneously. For more details on how relators fix ambiguities of material relations, please refer to (GUIZZARDI, 2005, chap. 6).

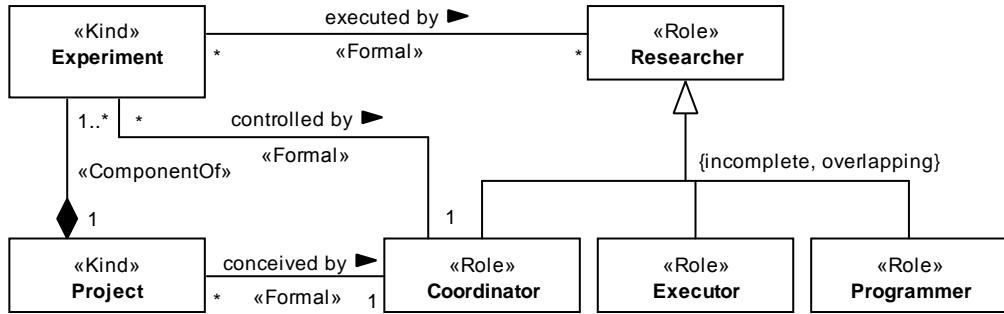


Figure 49. UndefFormal occurrence identified in the OVO ontology.

As a last remark about the UndefFormal anti-pattern, we recognize that the suggested identification structure is not the most efficient to point out improper uses of the formal relation. Even though a class has attributes, it does not mean that they participated in the definition of the formal relations connected to it. In the diagram presented in Figure 49, for example, if both the “*Researcher*” and the “*Experiment*” classes had an attribute “name”, the UndefFormal’s structure would miss the relations identified in the example.

5.20 UNDEFINED PHASE PARTITION (UNDEFPHASE)

Phases are anti-rigid types that aggregate individuals with the same identity principle and are instantiated due to an alteration on intrinsic properties. An intrinsic property can be either a quality, if it refers to a structured characteristic, like one’s age or weight, or a mode, if unstructured, like one’s headache, commitment or intention. Furthermore, phases are defined in partitions, i.e., they must be a part of a disjoint and complete generalization set.

Just like in the UndefFormal anti-pattern, the lack of intrinsic properties required to characterize a proper phase definition drives UndefPhase. It implies that a modeler did not specify the conditions necessary for an individual to instantiate a phase. Its structural definition is a partition of phases in whose common parent type does own or inherit attributes and associations connected to data types or modes.

One can think of many different situations that characterize a change in an intrinsic property. In general, an intrinsic property can appear or disappear, like diseases for example. Furthermore, we often expect that qualities to change their values through

the life cycle of an individual. Considering these possibilities, we propose two patterns for phase partitions: the derived partition and the intentional partition.

Derivation rules define all phases in a **derived partition** exclusively using values of intrinsic properties (modes and qualities). For example, suppose that a person's body fat percentage (a quality) classifies them in the following phases: Slim, for values from lower than 25%; and Fat, if the value is equal or greater than 25%. To model this pattern, one needs to specify at least one attribute, used to define all phases in the partition. Furthermore, one must set all phases as derived types, alongside with each derivation rule.

The **intentional partition**, conversely, does not require the common parent type of the partition to own or specify attributes. The appearance and disappearance of intrinsic properties characterize this pattern. An example is a partition containing the sick and healthy phases of a person. A sick person is a phase characterized by the mode disease, whilst the absence of a disease characterizes healthy. Furthermore, intentional phase partitions can optionally have one phase that is derived by exclusion (OLIVÉ, 2007) from the others, i.e., an individual instantiates it if, and only if, it is an instance of their direct super type and is not an instance of any other phase in the partition.

We provide a compact and structured description of the UndefPhase anti-pattern in Table 50.

Table 50. Characterization summary of UndefPhase the anti-pattern.

Name (Acronym)		Description
Undefined Phase Partition (UndefPhase)		A partition of phases in whose common parent type does own or inherit attributes and associations connected to data types or modes.
Type	Feature	Justification
Classification; Scope		
Phase		Phases are anti-rigid types that are instantiated due to an alteration in an intrinsic property (a quality or a mode). For that reason, if the parent type of a partition does not have any intrinsic properties, how does one expect to define a partition?
Pattern Roles		
Mult.	Name	Possible Types
1	SuperType	«kind», «quantity», «collective», «subkind», «phase» and «role»

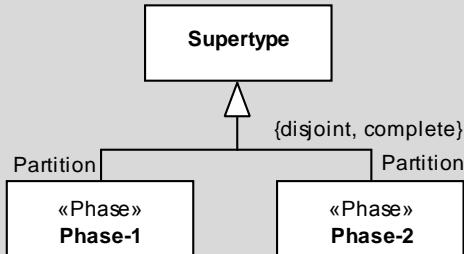
1	Partition	Generalization Set
2..*	Phase-n	«phase»

Additional Constraints

- Let qualities(c) be the function that return all qualities defined for a class c (through attributes or relations) and ancestor(c) be the function that return all direct and indirect super types of a class c, then:

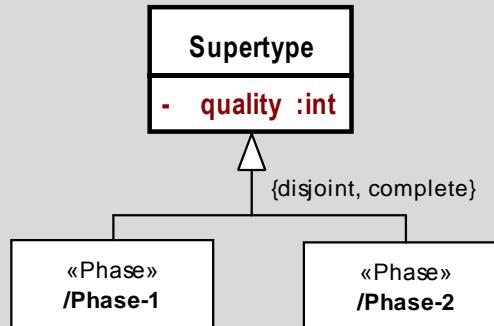
$$\#qualities(SuperType) = 0 \wedge \forall x \in ancestor(SuperType), \#qualities(x) = 0$$

Generic Example

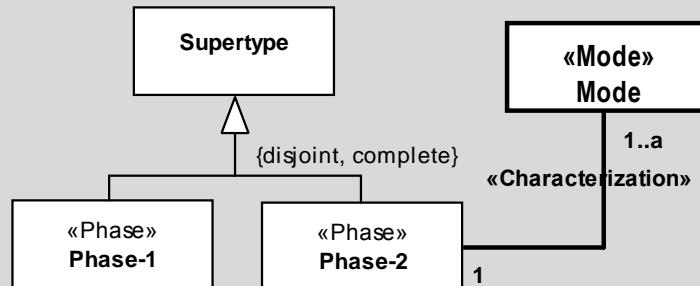


Refactoring Plans

- [New/OCL] Derived partition:** choose this option if the instantiation of the phases is defined by a change in a quality's value, owned by the common parent type, one of its ancestor, one of its parts or one of its modes. (e.g. Person-Adult-Child)



- [New] Intentional partition:** choose this option if the instantiation of the phases is defined by the appearance of a mode or a quality in the phases (e.g. Person-Sick-Healthy)



- [Mod/New] Set phases as roles:** choose this option if the instantiation of the phases is defined by a relational property and not an intrinsic one. To fix, change the stereotype of all phases to role and define their respective relational dependencies.

Anti-Pattern Relations

Group by Feature (Phase): DepPhase

Group by Type (Classification): DepPhase, GSRig, HetColl, HomoFunc, MixIden, MixRig, RelRig, UndefFormal

Group by Type (Scope): DepPhase, FreeRole, ImpAbs, MultDep, GSRig, HomoFunc, MixIden, MixRig, RelRig

Causes: none

Caused by: none

We exemplify UndefPhase with the two fragments depicted in Figure 50. In the left, we have a fragment of the MPOG Ontology Draft (MPOG, 2011), which describes different types of organizations that are legal in Brazil. In the right, an excerpt of the Health Organization Model, which describes concepts related to public health activities.

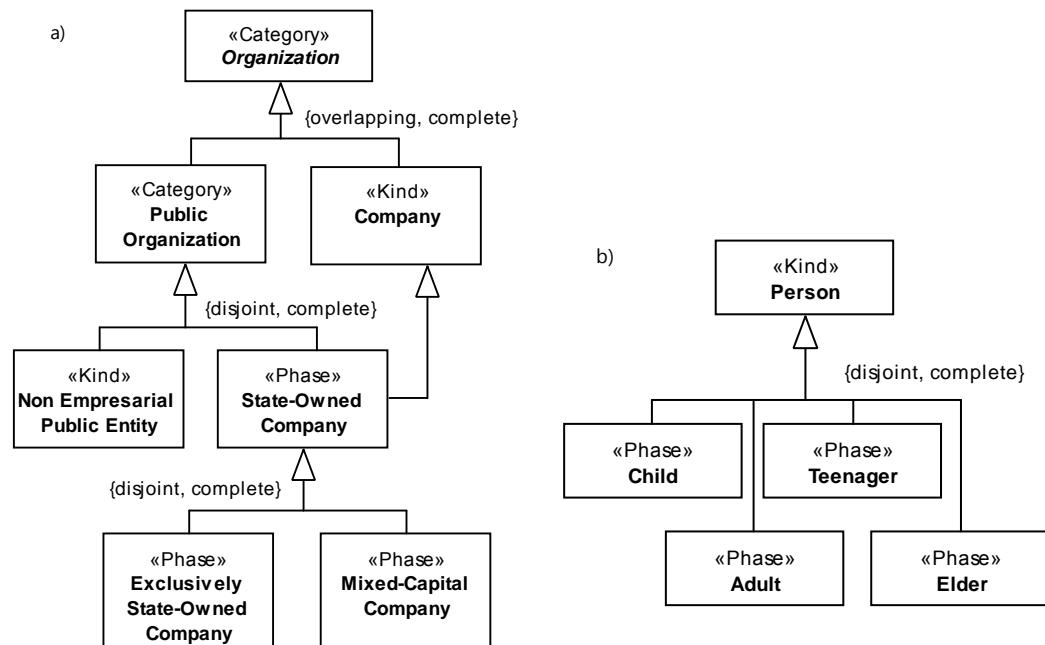


Figure 50. UndefPhase occurrences identified in a) the MPOG Ontology Draft and b) the Health Organization Model

In the MPOG model excerpt, the partition that characterizes the UndefPhase contains “*Exclusively State-Owned Company*” and “*Mixed-Capital Company*”. They are both types of companies owned by the government but in the first, the state is the single owner, whilst in the second, the state shares the control with the private sector. Petrobras, a Brazilian oil and gas company, has a mixed-capital. The British Broadcasting Company, widely known as BBC, is also a state-owned company. Notice

that the model itself does not define any properties for both phase, leaving the whole their whole meaning embedded in their labels. Using the available definition of the phases, we assume that they are not phases, but roles. What defines if a company is state owned or not is its constitution (state-owned are defined by some sort of law or regulation published by the government).

Now, analyzing the Health Organization excerpt, we immediately conclude that the partition fits the derived partition pattern. It requires the formalization of an “age” quality (possibly as an attribute of the integer type) and the definition of intervals for each phase, e.g. 0-12 is a Child, 13-17 is a Teenager, 18-64 is an Adult, and 65-* is an Elder.

5.21 WHOLE COMPOSED OF OVERLAPPING PARTS (WHOLEOVER)

The Whole Composed of Overlapping Parts (WholeOver) anti-pattern follows the exact same logic as RelOver. The main difference is that, instead of the focus being on a relator mediating overlapping types, it is on a whole type having overlapping parts. We refrain from providing redundant definitions and just present a WholeOver occurrence as an example. For more details, please refer back to RelOver’s definition in Section 0.

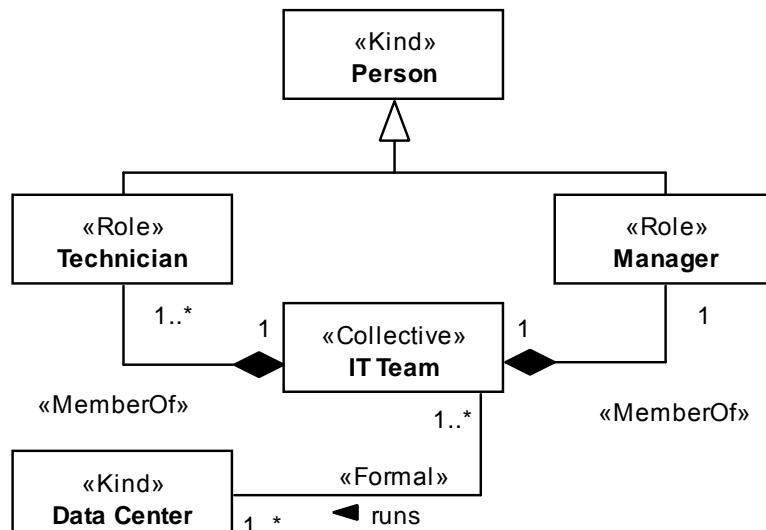


Figure 51. WholeOver occurrence identified in the IT Infrastructure model

Figure 51 presents an excerpt of the IT Infrastructure Model that contains a WholeOver occurrence, which follows from the whole class “*IT Team*” being composed by at least one “*Technician*” and exactly one “*Manager*”. The first inquiry to make is whether a Person can simultaneously be a Technician and a Manager, i.e., if the roles are indeed overlapping. If they ought to be disjoint, the creation of a generalization will fix the model. If they are really overlapping, the analysis continues by verifying if the parts are exclusive w.r.t the whole. If it is possible for the same person to be a manager and a technician in the same team no rule is necessary. If people can only play a single role in each instance of “*IT Team*”, the OCL rule in Listing 19 is in order.

Listing 19. Exclusive parts constraint defined in OCL.

```
context Person
inv: self.technicianoclAsType(Person) -> asSet() -> excludesAll(
  self.manageroclAsType(Person) -> asSet())
```

Lastly, we provide a summary of the WholeOver anti-pattern in Table 51.

Table 51. Characterization summary of the WholeOver anti-pattern.

Name (Acronym)		Description
Whole Composed of Overlapping Parts (WholeOver)		A whole composed of two or more types whose extension possibly overlap. The sum of the meronyms’ upper bound cardinalities of the part end must be greater or equal to 2 or at least one of them be unlimited.
Type	Feature	Justification
Logical		
Part-Whole		This structure is usually too permissive. It is often the case that some of the part types should be disjoint or set as exclusive in the context of a single whole instance.
Pattern Roles		
Mult.	Name	Possible Types
1	Whole	All object types: «kind», «collective», «quantity», «subkind», «phase», «role», «roleMixin», «category» and «mixin»
2..*	partOf-n	All meronymic stereotypes: «subQuantityOf», «componentOf», «memberOf», «subCollectionOf»
2..*	Part-n	All object types: «kind», «collective», «quantity», «subkind», «phase», «role», «roleMixin», «category» and «mixin»

Additional Constraints

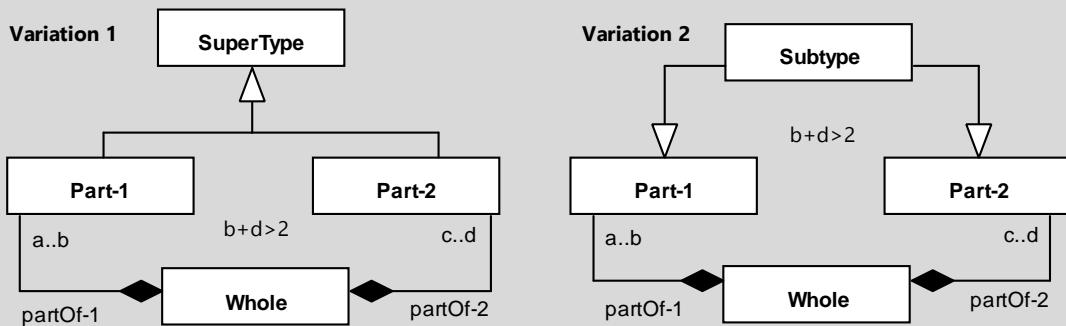
- Let M be the set of identified meronymic relations, $\text{partEnd}(m)$ the function that returns the association end connected to the part of a meronymic relation m , and $\text{upper}(p)$ the function that return the upper bound cardinality of a property p , then:

$$\left(\sum_{m \in M} \text{upper}(\text{partEnd}(m_n)) \right) \geq 2$$

- Let O be the set of part types that compose Whole, then:

$$\exists x, y \in O \mid \text{overlap}(x, y)$$

Generic Example*



*Note: the presented variations are illustrative and do not intend to cover all possibilities

Refactoring Plans

- [OCL] Exclusiveness:** choose this option to forbid the same individual to play multiple roles w.r.t the same whole instance. Create an OCL invariant according to the following template:


```
context Whole
inv:
self.part1.oclAsType(Supertype) ->asSet() ->excludesAll(
self.part2.oclAsType(Agent) ->asSet() and
self.part1.oclAsType(Supertype) ->asSet() ->excludesAll(
self.part3.oclAsType(Agent) ->asSet() and
self.part2.oclAsType(Supertype) ->asSet() ->excludesAll(
self.part3.oclAsType(Agent) ->asSet()))
```
- [OCL] Partially exclusiveness:** choose this option to set a subset of the part types as exclusive.
- [New/Mod] Disjoint parts:** Enforce part types to be disjoint through the creation or alteration of a disjoint generalization set.

*Note: to make all types exclusive, every binary combination should be explicitly ruled out

Anti-Pattern Relations

Group by Feature (Part-Whole): HetColl, HomoFunc, PartOver

Group by Type (Logical): AssCyc, BinOver, Declnt, FreeRole, ImpAbs, MultDep, PartOver, RelOver, RelComp, RelSpec, RepRel

Causes: none

Caused by: HomoFunc

5.22 PSEUDO ANTI-RIGID

In (SALES; BARCELOS; GUIZZARDI, 2012), we published the initial version of the anti-pattern catalogue, proposing 6 anti-patterns. Amongst them, there was one named Pseudo Anti-Rigid (PAR). Its proposal was to identify logical contradictions in the model by indicating anti-rigid classes (stereotyped as role, phase or roleMixin) forced into rigidity through external constraints. For example, if one defines a role class to represent the student type, one expects individuals to instantiate it in a moment and cease to do so in another, but keep existing. However, some models are so over-constrained that this possibility is precluded.

The structural causes for logical contradictions that hamper anti-rigidity, however, are so diverse that we did not identify a recurrent structure to characterize an anti-pattern. Nonetheless, we were still able to identify some structures that would cause such problems:

- an anti-rigid type characterized by an existential dependency, i.e., a role connected to an association whose opposite end has *isReadOnly* set to true;
- a generalization set of a rigid parent with at least one rigid subtype and exactly one anti-rigid subtype, as described in the GSRig anti-pattern; and
- mandatory closed instance level cycles composed by anti-rigid types, like the one described in (SALES; BARCELOS; GUIZZARDI, 2012) and depicted in Figure 52. Notice that a heart is necessarily composed of cells that always generate electrical impulses, which must provoke contractions that make an atrium work as a pump, which in turn makes the heart work as a pump. Assuming that the cells can only generate electrical discharges on hearts they are part of (the closed instance-level cycle) a heart always work as a pump.

In fact, we concluded that PAR presented a recurrent undesired consequence, but no recurrent structure. Considering that our definition for semantic anti-pattern requires an identifiable structure so we can prescribe pre-defined refactoring solutions, PAR did not fit. For that reason, we removed it from the catalogue.

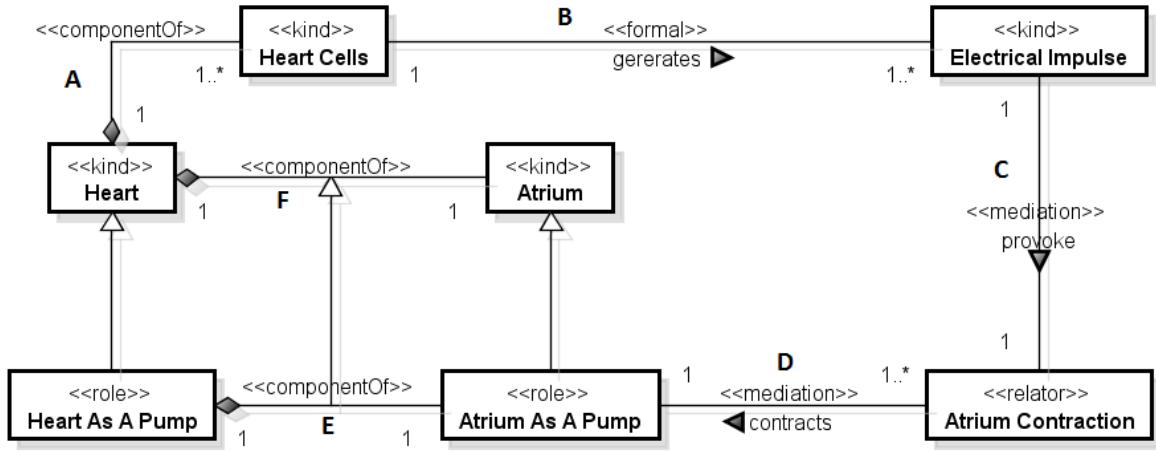


Figure 52. An ECG ontology's simplified excerpt illustrating PAR.

However, we did not cast it aside. PAR presents a recurrent problem that affects the instantiation of the model and the Alloy support for OntoUML is able to check assertions on a model. We then, transformed PAR into a standard test. It will not directly indicate to the modeler the cause of the problem, but it will let him know that there is one.

5.23 THE RELATOR DILEMMA

We refer to the other recurrent problem not classified as anti-pattern as the Relator Dilemma. Even though the dilemma suggests a classification issue, i.e. an error assigning a meta-category for a concept, we did not create an anti-pattern for it because we could not identify a recurrent structure. The only thing we could define, from a structural point of view, is that a relator is the source of the problem. In order to say that a relator class does not formalize a relator concept, we would need to analyze its label, which is out of the scope of this work.

We call it “The Relator Dilemma” because it an improper formalization of a concept as a relator is the source of the problem. Exploring the models in our repository, one finds all sorts different concepts classified as relators: complex and simple events (types of perdurants in UFO), actions, normative descriptions and even material relations.

Our hypothesis is that these errors arise from the modeler’s limit understanding of the relator and material association meta-categories, alongside with limited knowledge

about meta-categories not covered by the language, like events and normative descriptions. To exemplify the concepts of relator, event, normative description and material association we use the classical example of the marriage domain:

"James and Camille are a young couple that decided to get married. Since the bride came from a traditional family, they had a big wedding, with all their friends and family. After the wedding, Camille decided to take James' name, so they headed to the public records department, with their marriage certificate in hand. After proving their marital status, they manage to fix Camille's name and lived happily ever after!"

In our little tale, the wedding is the event that creates the relator, also referred to as the foundation of the relator. After the wedding, James and Camille can say that they are married to one another, thus, "being married to" is the material relation. The reason they can say that is because, after the wedding, a marriage that involves both of them arises: the relator. Now, if they need to prove that they are married to a third person, they just need to show their marriage certificate, i.e. the document that formalizes their married status: the normative description.

6 UNCOVERING SEMANTIC ANTI-PATTERNS

In this chapter, we present and discuss the methods we adopted to come up with the catalogue of semantic anti-patterns presented in the last chapter. We start by describing the repository of OntoUML models we used as a benchmark to elicit and validate the anti-patterns. In the sequence, we discuss the three complementary elicitation methods we adopted. First, the method based on model simulation, followed by the one based on the analysis of the foundational ontology, and finally the method driven by the comparison of OntoUML with other modeling languages. At the end of this chapter, we discuss the effectiveness of applying these methods and report on our experiences in doing so.

6.1 THE ONTOUML MODEL REPOSITORY

Throughout this research, we gathered all sorts of OntoUML models we could find in order to build a diversified model repository. Our goal was to have models with: (i) different domains; (ii) different levels of modeling expertise in Ontology-Driven Conceptual Modeling; (iii) models of different sizes, maturity and complexity; (iv) models developed for different purposes; and (v) different development contexts.

This repository serves two purposes in our research. First, as a source of inspiration for anti-pattern identification. We would not be able to propose anti-patterns if they were not identified in practice. The complementary utility of the model repository is for validation. By having access to the documentation of the ontologies and the modelers that designed them, we can evaluate if our proposals indeed capture problematic modeling decisions.

6.1.1 Ontology Template

We describe each of the 54 models in the repository according to the following template:

- **Name:** the name provided by the authors. If they did not provide any, we baptize the model with an intuitive one.
- **Domain:** a brief summary of the domain captured by the ontology, accompanied by examples of classes and properties represented in the ontology.
- **Context:** the scenario in the model was developed. One of the following classifications is given:
 - *Academy*, which include models produced in the context of academic research and described in bachelor thesis (BSc), master thesis (MSc), doctoral thesis (PhD) or research papers (Paper);
 - *Industry*, which classifies models produced in cooperation with / by the industry organizations;
 - *Government (Gov)*, which classifies models developed by or in cooperation with governmental entities;
 - *Graduate Course Assignment (GCA)*, encompassing models produced by graduate students as a final assignments of an “Ontology Engineering” 60-hour course offered by the Graduate School on Informatics of the Federal University of Espírito Santo; and
 - *others, for models that don't fit in any of the aforementioned categories*
- **Type:** binary parameter that classifies the model in either *ontology* or *conceptual model*:
 - *Ontology (Ont)* is assigned to models that are meant to capture the conceptualization of a community, and
 - *Conceptual Model (CM)* classifies models that are not truly ontologies, in the meaningful sense of the word, since one can hardly say they represent a consensus of a community. Mostly, they are formalizations of a single modeler or a very small group.
- **Purpose:** describes the motivation to build the ontology, the problem the authors designed it to solve. The categories applied to this field are:

- *interoperability*, for the ones that were meant to be used as reference for semantic interoperability between agents and/or systems within the same of multiple organizations;
 - *ontological analysis (onto analysis)*, for the models which were developed to evaluate conceptual modeling languages or ontologies produced in other languages;
 - *reference model (reference)* for those models that are proposed as references for a community;
 - *knowledge-based application (kb application)* to classify those that are created during the development of knowledge-based applications; and
 - *unspecified*, for the ones which the author did not provide any application;
- **Expertise (Exp.)**: describes the author's familiarity to the OntoUML language at the time of the development of the ontology. We assign to values for this field:
 - *Beginner (beg)*, for modelers with less than a year of experience using the language; and
 - *Advanced (adv)*, for modeler who use the language for more than 2 years.
 - **Modelers (#Md)**: provides the number of modelers that participated in the development of the ontology. For published models, we consider all authors of the publication as modelers. For models developed in thesis, we consider the student and the supervisors.
 - **Structural Data**: describes the model from a quantitative structural perspective, providing the number of classes, associations, attributes, generalizations, generalization sets, datatypes and packages.

Repository Overview

Table 52 provides a general description of all ontologies in the repository, following the aforementioned template. From the 54 models, 18 (33%) are ontologies in the proper sense, i.e., actual formalizations of shared conceptualizations. The remainder 36 (67%) are just regular conceptual models that capture a modeler's perspective about a portion of reality, which may be shared by others or not.

Analyzing the development context, we can see that most models are graduate course assignments, a total of 32 or 59% of the repository. Another 11 (20%) are models developed in academic researches without industry collaboration. An example is The Configuration Management Task Ontology (CMTO) (CALHAU; FALBO, 2012), a product of a Masters research. Moreover, seven models had the total or partial participation of private companies and/or governmental organizations. The most significant one is the MGIC Ontology (BASTOS et al., 2011), developed in the context of a research project with a Brazilian regulatory agency, named “*Agência Nacional de Transportes Terrestres*⁵” (ANTT), which is responsible for regulating ground transportation in Brazil. Only the development context of four models is unknown or was not available.

Concerning the purpose to which the models were created, the repository contains 10 models (16%) that are intended to serve as a reference knowledge of a domain, like UFO-S (NARDI et al., 2013). Another 10 models (16%) were developed in order to perform ontological analysis on existing models, databases or modeling languages. An example is the refactoring of the Conceptual Schema of Human Genome presented in (FERRANDIS; LÓPEZ; GUIZZARDI, 2013). The repository also contains eight models (13%) designed for the development of knowledge-based applications, 6 (10%) whose main intention was to support semantic interoperability between systems and/or organizations, and only two (3%) for enterprise modeling. For the remainder 26 models (42%), we do not know the motivation or the authors did not inform it. Notice that since most of these “purposeless” models are also graduate course assignments, it does not raise any further questions.

Concerning the modeler’s overall expertise on OntoUML and Ontology-driven Conceptual Modeling, we identify 22 models (41%) developed by beginners (from which 18 are also graduate course assignments) and 32 (59%) developed by experienced modelers.

Finally, we look into the number of modelers involved in the ontology design. A single person participated in the development most of the time (35 models, roughly 65%). Moreover, fifteen models are the product of collaboration efforts between 2-4 people,

⁵ In English: National Ground Transportation Agency

whilst four involved 7-10 ontologists. By comparing the number of modelers and the development context features, we identify that, from the 35 developed by a single modeler, 31 are graduate course assignments.

Table 52. Summary description of all models in the repository.

Model	Type	Domain	Context	Purpose	Exp.	#Md
The MGIC Ontology	Ont	Brazilian Ground Transportation Regulation	Gov	interoperability; enterprise md.	adv	10
The G.805 Ontology	Ont	ITU-T G.805 Recommendation	Industry	onto. analysis; reference	adv	4
The G.805 Ontology 2.0	Ont	ITU-T G.805 Recommendation	Industry	reference; kb application	adv	3
The G.800 Ontology	Ont	ITU-T G.800 Recommendation	Industry	reference; kb application	adv	3
OntoEmergePlan	Ont	Emergency Plans	MSc	kb application	adv	8
OntoUML Org Ontology	Ont	Enterprise architecture	MSc	onto. analysis; enterprise md.	adv	2
The ECG Ontology	Ont	Eletrocardiogram	MSc	interoperability; kb application	adv	3
Gi2MO Ontology Refactored	CM	Generic idea and innovation management	GCA	onto. analysis	adv	1
Internal Affairs Ontology Refactored	CM	Brazilian police internal affairs dept.	GCA	onto. analysis	adv	3
OVO	Ont	Provenance of scientific experiments	PhD	reference	adv	3
The Library Model	CM	Library archive and services	GCA	unspecified	beg	1
OntoBio	Ont	Amazonian biodiversity	Gov	interoperability; kb application	adv	3
The Public Tenders Model	CM	Brazilian public tenders	GCA	unspecified	adv	1
UFO-S	Ont	Commitment-based Service	PhD	reference	adv	7
The TM Forum Model	CM	Network management	Other	unspecified	beg	1
The Social Contract Model	CM	Brazilian social contract theory	GCA	unspecified	beg	1
The Clergy Model	CM	Catholic clergy	GCA	unspecified	beg	1
The FIFA Football Model	CM	Football (based on FIFA's offical rules)	GCA	unspecified	adv	1
The PAS 77:2006 Ontology	Ont	Service continuity	MSc	onto analysis; reference	adv	4
IDAF Model	CM	Institute of Agriculture Protection of Espírito Santo	GCA	unspecified	adv	1
The Cloud Vulnerability Ontology	Ont	IaaS perspective on public cloud vulnerability	MSc	reference	adv	4
The University Model	CM	Brazilian federal universities	GCA	unspecified	beg	1
CMTO	Ont	Configuration Management of Software Products	MSc	interoperability	adv	2
GRU MPS.BR Model	CM	Reuse management process of MPS.BR	GCA	unspecified	beg	1
The Experiment Model	CM	Scientific experiment	GCA	unspecified	beg	1
CSHG Refactored	Ont	Human genome	Paper	onto analysis	adv	3

The Normative Acts Ontology	Ont	Brazilian normative acts composition	Gov	reference	adv	3
The Parking Lot System	CM	World view of a parking lot management system	GCA	unspecified	adv	1
The School Transportation Model	CM	World view of a system to support student transportation	Other	application	beg	1
The Quality Assurance Ontology	CM	Quality assurance based on CMMI, MPS.BR and ISO 9001	GCA	unspecified	adv	1
The OpenFlow Ontology	Ont	OpenFlow communication protocol	BSc	kb application	beg	2
The Music Ontology Refactored ⁶	CM	Music-related data	GCA	onto analysis	adv	1
The Internship Model	CM	Legal brazilian internship	GCA	unspecified	adv	1
The G.809 Model	CM	ITU-T G.809 Recommendation	GCA	unspecified	beg	1
The ERP System Model	CM	World view of an enterprise resource planner system	Other	interoperability	adv	1
The Online Mentoring Model	CM	World view of a system to support online mentoring	GCA	unspecified	adv	1
The Help Desk System Model	CM	A model that describes a potential help desk system	GCA	unspecified	beg	1
The IT Infrastructure Model	CM	Information technology architecture	GCA	unspecified	beg	1
The Requirements Ontology	Ont	Software requirements	MSc	kb application	adv	2
The Photography Model	CM	Photography collection	GCA	unspecified	beg	1
FIRA Ontology Refactored	CM	Robot soccer matches	GCA	onto analysis	adv	1
The Banking Model	CM	Financial operations	GCA	unspecified	adv	1
The Chartered Service Model	CM	Railway chartered service	GCA	unspecified	adv	1
The Health Organization Model	CM	Brazilian health organization	GCA	unspecified	beg	1
The Bank Model 2	CM	Financial operations	GCA	unspecified	adv	1
The PROV Ontology Refactored ⁷	CM	Provenance information	GCA	onto analysis	beg	1
WSMO Refactored ^{8,9}	CM	eGovernment services	GCA	onto analysis	beg	1
The Rec. Model	CM	Recommendations and norms	GCA	kb application	beg	1
The Inventory System	CM	World view of an inventory management system	Other	interoperability	adv	1
MPOG Ontology Draft	Ont	Brazilian federal organizational structures	Gov	reference	beg	7
The Project Management Model	CM	Project management	GCA	unspecified	beg	1
The Construction Model	CM	Construction	GCA	unspecified	beg	1
The Stock Model	CM	Stock brokers	GCA	unspecified	beg	1
The Real State Model	CM	Real state	GCA	unspecified	beg	1

Table 53 provides a structural overview of the repository. Notice that there are ontologies of all sizes, from enormous ones (e.g. The MGIC Ontology, with 3800

⁶ OWL ontology used to create the OntoUML version available at: <http://musiconontology.com/>

⁷ OWL ontology used to create the OntoUML version available at: <http://www.w3.org/TR/prov-o/>

⁸ WSMO: Web Service Modeling Ontology

⁹ OWL ontology used to create the OntoUML version available at: <http://www.wsмо.org/>

classes and 1918 associations), to medium (e.g. OntoUML Org Ontology, containing 78 classes and 78 associations), to very tiny ones (e.g. The Chartered Services Model, formalizing 11 classes and 14 associations).

Furthermore, Table 53 provides information of the number of data types (includes complex datatypes, enumerations and primitive types), generalizations (Gen.), generalization sets (GS) and attributes (Attr.) in each model.

Table 53. Structural description of all models in the repository

Model	Class	Datatype	Assoc.	Gen.	GS	Attr.
The MGIC Ontology	3800	61	1918	3616	698	865
The G.805 Ontology	135	4	113	127	36	0
The G.805 Ontology 2.0	358	1	255	475	62	7
The G.800 Ontology	477	1	345	601	78	7
OntoEmergePlan	189	4	138	111	16	5
OntoUML Org Ontology	78	0	78	57	8	0
The ECG Ontology	49	0	65	31	0	0
Gi2MO Ontology Refactored	65	5	63	42	7	2
Internal Affairs Ontology Refactored	62	0	54	36	9	0
OVO	49	0	50	26	4	0
The Library Model	43	0	45	14	0	0
OntoBio	187	5	50	160	22	14
The Public Tenders Model	84	0	43	48	6	18
UFO-S	22	0	42	4	0	0
The TM Forum Model	34	0	41	20	4	0
The Social Contract Model	20	0	15	16	0	2
The Clergy Model	29	0	34	16	0	0
The FIFA Football Model	68	1	32	69	4	2
The PAS 77:2006 Ontology	66	0	32	55	11	0
IDAF Model	46	0	32	38	0	0
The Cloud Vulnerability Ontology	33	0	29	21	2	0
The University Model	27	4	29	16	0	0
CMTO	41	0	28	28	0	0
GRU MPS.BR Model	19	7	28	15	3	18
The Experiment Model	20	2	26	0	0	0
CSHG Refactored	19	0	22	10	1	0
The Normative Acts Ontology	63	1	21	55	17	24
The Parking Lot System	49	0	21	37	9	17
The School Transportation Model	33	0	36	9	0	0
The Quality Assurance Ontology	41	0	20	24	7	2
The OpenFlow Ontology	20	0	19	9	1	4
The Music Ontology Refactored	43	0	18	36	6	5
The Internship Model	26	6	18	19	4	2
The G.809 Model	24	0	18	12	4	0
The ERP System Model	38	0	16	25	1	43
The Online Mentoring Model	29	0	16	18	6	0

The Help Desk System Model	20	0	16	8	4	0
The IT Infrastructure Model	31	0	15	17	6	0
The Requirements Ontology	35	1	21	30	10	19
The Photography Model	19	0	15	8	0	0
FIRA Ontology Refactored	41	0	14	36	7	0
The Bank Model	18	0	12	14	4	2
The Chartered Service Model	11	0	14	0	0	0
The Health Organization Model	24	0	13	14	3	0
The Bank Model 2	24	1	14	16	3	3
The PROV Ontology Refactored	16	0	12	5	0	0
WSMO Refactored	12	0	12	2	0	0
The Rec. Model	16	0	10	11	3	6
The Inventory System	20	0	7	14	0	24
MPOG Ontology Draft	15	0	7	15	4	0
The Project Management Model	14	0	7	8	3	0
The Construction Model	13	0	7	7	0	0
The Stock Model	14	0	6	11	7	0
The Real State Model	15	0	5	13	0	0

In Appendix B, we elaborate on the most noteworthy ontologies, namely the ITU-T ontologies, OntoEmergePlan, OntoBio and OntoUML Org Ontology. For each, we provide further information about the domains, applications and the stereotype usage in the model.

6.2 SIMULATION-BASED INVESTIGATION

Throughout this research, we employed different approaches for the identification of anti-patterns in OntoUML conceptual models. The first and most used was an empirical qualitative analysis. The idea was to simulate existing models (using the methods and tools described in Chapter 3) and identify recurrent modeling problems. To do that, we needed to gather as many models as possible with the most characteristics variation as possible.

Figure 53 depicts a flowchart of the protocol adopted in our studies for the identification of new anti-patterns. For each model under analysis, we started by dividing it into smaller fractions, then simulating these model fragments using the approach described in Chapter 3. This process resulted in a number of possible model instances for that model (automatically generated by the Alloy Analyzer). We then contrasted the set of

possible instances with the set of intended instances of the model, i.e., the set of model instances that represented intended state of affairs according the creators of the models. When we detected a mismatch between these two sets, we analyzed the model in order to identify which structures (i.e., combination of language constructs) were the causes of such a mismatch. We then proceed to fix the model and register the combination of problem and solution.

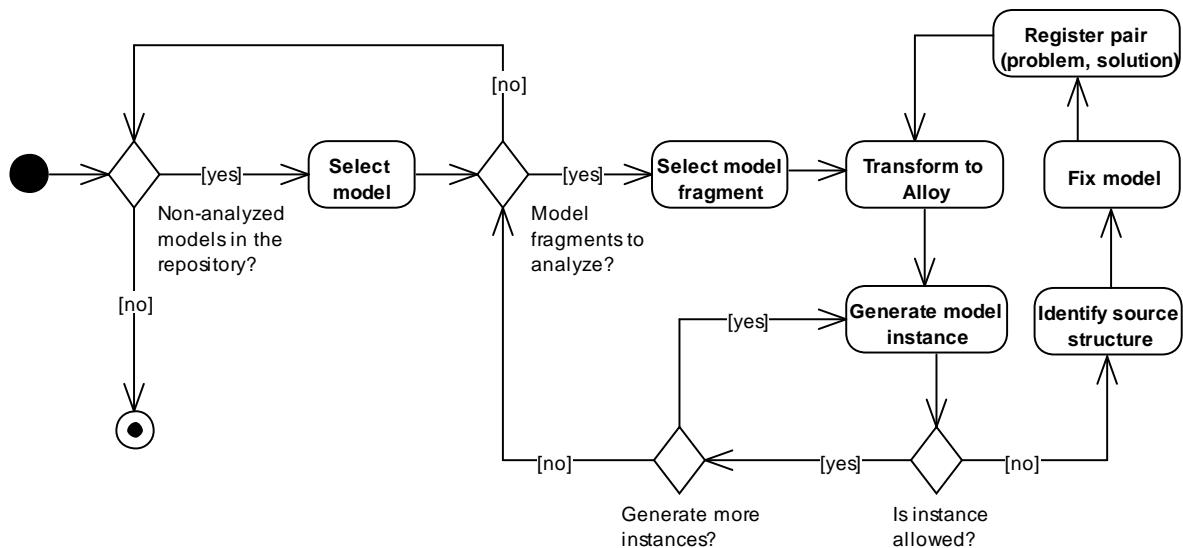


Figure 53. Detailing of the protocol adopted for anti-pattern identification through visual simulation

Finally, after analyzing the models, we catalogued as anti-patterns those model structures that recurrently produced such domain misrepresentations, i.e., modeling patterns that would repeatedly produce model instances, which were not intended ones, or that would not produce the expected model instances. We carried out this simulation-based validation process with a constant interaction with the model creators (when available), or by inspecting the textual documentation accompanying the models.

To conclude, we point out that we designed the aforementioned protocol to identify two types of problems: under and over constraining. This expectation lies on the fact that we use a simulation-based approach. When we identify an undesired possibility, we say that the model is under-constrained. When we expect a model instance, but we do not encounter it, the model is over-constrained. For this reason, we only encountered Logical and Scope Anti-Patterns (see Section 4.2) with this approach.

6.2.1 The First Empirical Study

The preliminary study presented in this section is published in (SALES; BARCELOS; GUIZZARDI, 2012).

The study was conducted with only 9 models, namely: The Health Organization Model, The University Model, The Online Mentoring Model, The G.805 Ontology, OntoBio, The ECG Ontology, The Normative Act Ontology (although during that time it wasn't published yet), The Public Tenders Ontology (a subdomain of the MGIC ontology) and The MPOG Ontology Draft. For more details about these models, please refer back to Table 52.

The study allowed us to identify six initial semantic anti-patterns, namely: *Generic Cycle* (now renamed to *Association Cycle*), *Relation Specialization*, *Imprecise Abstraction*, *Pseudo Anti-rigid*, *Type-Self Relationship* and *Relation Between Overlapping Subtypes* (in this thesis, we merged these last two into *Binary Relation With Overlapping Ends*).

Table 54. A summary of the results in the first study

Ontology	#GC	#RBOS	#RS	#IA	#TRR	#PAR
The Health Organization Model	1	1	0	1	0	0
The University Model	1	1	1	3	0	0
The Online Mentoring Model	3	2	0	1	0	0
The G.805 Ontology	9	1	3	3	4	1
OntoBio	2	2	11	3	3	0
The ECG Ontology	2	0	2	2	0	2
The Normative Acts Ontology	8	3	0	3	0	0
The Public Tenders Ontology	2	4	1	0	0	0
MPOG Ontology Draft	2	0	2	1	2	1
Total	30	14	20	17	9	4
Percentage	100%	77.7%	66.7%	88.9%	33.3%	33.3%

A particular characteristic of this study is that we performed the protocol in every one of the nine models aforementioned. Furthermore, after came up with the set of potential anti-patterns, we manually inspect every model for occurrences and analyzed whether or not an occurrence characterized an error. Table 54 summarizes the number of occurrences (#) for the following anti-patterns in each of the investigated ontologies:

Generic Cycle (GC), Relation Between Overlapping subtypes (RBOS), Relation Specialization (RS), Imprecise Abstraction (IA), Type-Reflexive Relationship (TRR) and Pseudo Anti-Rigid (PAR). The number presented for each anti-pattern only encompasses those that were the cause of domain misrepresentations.

6.2.2 The Second Empirical Study

The initial study described in the previous section, gave us confidence that we could adopt the method as means for detecting these semantic anti-patterns. We then conducted a broader study, published in (SALES; GUIZZARDI, 2014). In this new study, our model benchmark contained not nine, but 52 models. The whole model repository listed in Section 6.1, except “The G.805 Ontology 2.0” and “The G.800 Ontology”.

Due to human resource limitations, we would not be able to conduct the study in the same way as the first. Therefore, in order to analyze this new benchmark, we implemented a set of computational strategies to automatically detect occurrences of these anti-patterns in OntoUML models (see discussion in Section 8.3). By running these algorithms for our initial set of anti-patterns under this benchmark, we managed to refine and extend the initial set elicited in (SALES; BARCELOS; GUIZZARDI, 2012) to a refined set of anti-patterns.

Table 55 reports on the results of this second study, which allowed us, at the time, to refine our anti-pattern catalogue. The anti-pattern named *Binary Relation Between Overlapping Types* is a refinement and combination of the previous *Self-Type Relationship* and *Relation Between Overlapping Subtypes* anti-patterns identified in the first study. Moreover, we elicited two additional anti-patterns in this second study, namely *Repeatable Relator Instances*¹⁰ and *Relator Mediating Overlapping Types*¹¹. Furthermore, the *Association Cycle* is the previously called *Generic Cycle*. Finally, we exclude the *Pseudo-AntiRigid* (PAR) anti-pattern, identified in our original catalog, from

¹⁰ In REF, the Repeatable Relator Instances (RepRel) anti-pattern was entitled Twin Relator Instances (TRI)

¹¹ In REF, the Relator Mediating Overlapping Types (RelOver) anti-pattern was entitled Relator With Overlapping Roles (RWOR)

the analysis conducted in this study. We did it because we could not specify an algorithm to identify its occurrences automatically.

Unlike in our first study (SALES; BARCELOS; GUIZZARDI, 2012), we were not able to check whether every occurrence indeed characterized a modeling error (3612 occurrences!). For this reason, the results reported in Table 55 stand for identified occurrences, regardless whether they are errors or not.

Table 55. Results of the second anti-pattern empirical study.

Semantic Anti-Pattern	% of models in which the anti-pattern occurs	Number of Occurrences in Benchmark
Relation Specialization	46.15%	1435
Imprecise Abstraction	71.15%	725
Association Cycle	51.92%	155
Relator Mediating Overlapping Types	30.70%	437
Repeatable Relator Instances	55.77%	685
Binary Relation Between Overlapping Types (incl. RBOS and STR)	48.07%	175
Total		3612

Other anti-patterns identified using the same method applied in this second study, but not published in (SALES; GUIZZARDI, 2014) and are, thus, first being presented in this thesis are: Whole Composed of Overlapping Parts, Part Composing Overlapping Wholes, Relator Mediating Rigid Types, Relation Composition, Multiple Relational Dependency and Free Role Specialization.

6.3 FOUNDATIONAL-BASED INVESTIGATION

The simulation-based approach has shown to be very promising. We, however, did not limit ourselves to uncover anti-patterns it exclusively. To improve and increase our anti-pattern catalogue, we sought inspiration in UFO, the foundational ontology that is in the core of OntoUML.

As we discussed in Chapter 2, UFO is the source of all stereotypes and constraints included in OntoUML's syntax. With that in mind, we took a closer look into the theories and definitions that explained the meta-categories of the language. Then, we proceed

to verify in our repository if the modelers used the meta-categories in the way the foundational ontology describes them. We were not looking for syntactical errors, but for valid modeling decisions, which are somehow uncommon, unexpected or even unorthodox.

To clarify what type of problem we were looking for, consider OntoUML's *Formal* stereotype. Guizzardi defines it in (2005) as a relation reducible to the comparison of quality values of the related individuals. In fact, Guizzardi refers to this concept as Comparative Domain Formal Relation. For example, the relation "older than" is formal, because it is reducible to the comparison of the related individuals' ages. In other words, it is true that Peter is older than John if and only if Peter's age is greater than John's age. Now, analyzing OntoUML's metamodel, one can see that there is no restriction involving the use of formal relations. This "freedom" allows modelers to relate whatever classes they want using the formal stereotype. The question we would investigate, thus, is: *are modelers using formal relations with the meaning of comparative domain formal relation?* If they are, are they accurately defining its semantics (providing the quality characteristics and the derivation rule)? If not, what are they meaning?

After we identified this semantic variation "gap" between the foundational ontology and the modeling language, we proceed to investigate manually a few models. If we identified a recurrent problem, we proceeded to implement a computational strategy to identify automatically the potential anti-pattern. Following, we would check our repository for occurrence and analyze samples to verify if it matched our expectation.

Investigations following the aforementioned process lead us mostly to the identification of what we called Classification Anti-Patterns, but we also found some Scope Anti-Patterns as well (please refer back to Section 4.2 about details on anti-pattern types). This particular type of semantic anti-pattern highlights possible errors in the decision making process applied to choose the appropriate modeling construct for a given concept.

A total of 9 anti-patterns were identified using this foundational ontology approach, namely: *Relationally Dependent Phase*, *Undefined Phase Partition*, *Undefined Domain Formal Relation*, *Mixin With Same Rigidity*, *Mixin With Same Identity*, *Generalization*

Set with Mixed Rigidity, Homogeneous Functional Complex, Heterogeneous Collection and Event x Truth-Maker x Normative Description. We discuss each anti-pattern in the next chapter.

6.4 INVESTIGATION BY COMPARISON

The second complementary approach we adopted consisted in the investigation of other modeling languages, like Object Role Modeling (ORM) (HALPIN; MORGAN, 2008), and compare them to OntoUML. We conducted this investigation in a much more informal way than the first two methods. We opted not to make formal comparisons, but instead, to seek inspiration. We sought modeling constructs that existed in other modeling languages but did not in OntoUML. Considering that OntoUML is the only ontology-driven conceptual modeling language so far, we did not expect to find anti-patterns regarding ontological distinctions. In fact, we aimed to uncover syntactical features that could point to constraint patterns to impose on the model.

UML is undoubtedly the most widespread modeling language, but in our case, it made no sense to investigate it because OntoUML is already an extension of UML's Class Diagram, so it inherited all UML modeling constructs. We decided, then, to inspect ORM (HALPIN; MORGAN, 2008) and it turned out to be quite productive. ORM's internal uniqueness constraint actually inspired the creation of the Repeatable Relator Instances anti-pattern, as well as its refactoring alternatives. The external uniqueness constraint influenced the Multiple Relational Dependency. Finally, the ring constraints indicated possible refactoring alternatives for the Binary Relation Between Overlapping Types. We will provide detail about ORM's influence on the anti-pattern catalogue as we present them in the next chapter.

6.5 CONCLUSION

In general, all three approaches we adopted for anti-pattern identification produced successful results, as evidenced by the 22 anti-patterns presented in the next chapter.

Nonetheless, we acknowledge that the empirical protocol presented in Section 6.2 has a few limitations, namely:

- how to divide the model into smaller fractions;
- how to generate relevant model instances; and
- the problem of identifying when a model instance characterizes a mistake.

The Alloy Analyzer, the tool we use to generate model instances, imposes the first issue, the need to identify small model fragments to simulate. However, even if we were able to simulate a big model and generate many instances, we have limited cognitive capacity, so it would be hard to analyze all together. Regardless, we are aware that the way one divides a model influences the errors one will find (or not).

The second issue regards the generation of model instances and it affects the performance of executing the protocol. The Alloy Analyzer generates all valid instances that fit a given specification. The problem is that the number of possible instances increases exponentially with the model. The task of generating instances with differently enough properties can be very laborious.

The third issue is the problem of identifying if a model instance is wrong or not. The way we did it, by visual inspection, is obviously error prone. We do not envision any alternative, so we had to accept that.

The last remark we make is that, even though we analyze the models in the benchmark looking for “bad” decisions, we encountered various domain dependent and independent positive modeling patterns. It was not the scope of this work to investigate these decisions, but the model benchmark we assembled is surely useful for that purpose.

7 EVALUATING THE ANTI-PATTERN CATALOGUE

We defined semantic anti-patterns as being error-prone recurrent modeling decisions. Therefore, in order to evaluate how useful the proposed anti-pattern catalogue is, we analyzed two characteristics of each anti-pattern: **frequency**, that measures how recurrent these modeling decisions are; and **accuracy**, that indicates how error-prone they are. In this section, we present the results of the two studies we conducted to individually assess each of these aspects.

In both studies, we evaluate the results individually (focusing on a single anti-pattern) and for the catalogue as a whole, aggregating results for all anti-patterns.

To conduct the experiments we used two software tools:

- Sparx Enterprise Architect (EA)¹², a commercial UML-based modeling tool that we used to re-construct the models we gathered (to models were specified using the OntoUML MDG plugin (SOBRAL; GUERSON; SALES, 2012)); and
- OntoUML Lightweight Editor (OLED)¹³, to automatically detect and analyze occurrences of the anti-patterns (for more details, see Chapter 8).

7.1 FREQUENCY EVALUATION

The goal of the first study is to evaluate anti-pattern **frequency**, i.e., how likely are modelers to design structures that fit an anti-pattern definition. Frequency is the first pillar on anti-pattern relevance: anti-patterns that are more common have a greater impact on ontology development, because it means that modelers often make those “dangerous” decision and this, more people can learn and take advantage from it. Remember that, even though an anti-pattern occurrence does not imply a mistake, in this first study, we were only interested if the structure defined by an anti-pattern is recurrent or not.

¹² <http://www.sparxsystems.com.au/>

¹³ <https://code.google.com/p/ontouml-lightweight-editor/>

To perform this evaluation, we adopted a fully automatic approach: we used our anti-pattern management tool developed for OLED (see Chapter 8) to identify automatically occurrences of each anti-pattern in each of the 54 models in our repository (see Section 6.1). We then aggregate the results and compared to some model characteristics to better characterize them.

The first variable we investigated is the sum of all anti-pattern occurrences in all models, which is useful to provide a dimension of the size of the population under analysis. Table 56 details the results, indicating the number of occurrences, the percentage regarding all occurrences and the anti-pattern type. Overall, we identified 6004 occurrences in the 54 models. Furthermore, we identified at least one instance of all anti-patterns.

Table 56. Summary with all identified occurrences in all models.

Anti-Pattern	All Occurrences	Occurrences/Total	Type
AssCyc	1809	30.13%	Logical
RelSpec	817	13.61%	Logical
ImpAbs	758	12.62%	Logical; Scope
RelComp	739	12.31%	Logical
RepRel	319	5.31%	Logical
UndefFormal	293	4.88%	Classification
RelRig	282	4.70%	Logical; Scope
BinOver	224	3.73%	Logical
RelOver	149	2.48%	Logical
HomoFunc	142	2.37%	Classification; Scope
FreeRole	119	1.98%	Logical; Scope
MultDep	105	1.75%	Logical; Scope
DeclInt	92	1.53%	Logical
HetColl	60	1.00%	Classification
WholeOver	27	0.45%	Logical
GSRig	16	0.27%	Classification; Scope
MixIden	16	0.27%	Classification; Scope
PartOver	13	0.22%	Logical
UndefPhase	11	0.18%	Classification; Scope
DepPhase	7	0.12%	Classification; Scope
MixRig	6	0.10%	Classification; Scope
Total	6004	100%	

We organized the table by the number of occurrences, in a decreasing order. Thus, the most identified anti-pattern is AssCyc, which corresponds to a significant 30% of

all anti-patterns. RelSpec, ImpAbs and RelComp follow it, all with more than 700 occurrences. We expected these results, since these are the only stereotype-independent anti-patterns.

Now, if we aggregate the results by anti-pattern type, we get that 5348 are logical in nature, whilst 1462 indicate scope issues and only 551 point to classification problems. These numbers are reasonable if we take into account the fact that we identified the logical anti-patterns through empirical analysis, whilst we proposed the classification ones from theoretical studies.

The variability of anti-pattern occurrences per model is so great that the average does not provide an insight on how many occurrences per model we should expect. AssCyc, for example, has 33 occurrences as average and standard deviation of 126, a variation ration of almost 400%.

The sum of occurrences is not the only dimension adopted to characterize anti-pattern frequency because it does not show the distribution in different models. A high number of occurrences in one model, for example, might hide the fact that many models register no occurrence. To cope with that, we describe in Table 57, the number of models with at least one occurrence of a given anti-pattern, alongside the corresponding percentage if compared to all models in the repository. This provides us indication on how many different modelers produced anti-pattern occurrences.

In a decreasing order, the table shows that again, AssCyc is the most frequent anti-pattern. Not only has it occurred many times, but in many different models. From the 54 for investigated model, 50 registered at least one occurrence of AssCyc, what corresponds to 92.59% of the repository. This results show that some stereotype-specific anti-patterns, like RelRig, RepRel, MultDep (all defined around relators and mediations), and UndefFormal (specified using formal associations) move up in the frequency list.

We identified almost all anti-patterns in multiple models, an indication that they are indeed recurrent modeling decisions. Particularly, we identified AssCyc in 93% of the models. We identified five anti-patterns in 50-75% of the models, another 5 in 25-50% of the models and 10 in 25% or less models.

We only identified two anti-patterns in a single model, namely PartOver and MixRig. In both cases, we only identified occurrences in the MGIC ontology. PartOver is an anti-pattern derived from RelOver (in the same way that WholeOver was), so if we consider all overlapping anti-patterns together, we found 189 occurrences in 16 different models. In addition, MixRig is an anti-pattern that indicates an improper use of the mixin concept. The identification of MixRig in only one model is explained by the seldom use of the mixin stereotype. In fact, in the whole repository, only seven models use it. Therefore, one out of seven is not that bad, since it corresponds to 14% of the models.

Table 57. Anti-Pattern frequency on investigated models.

Anti-Pattern	Models with Occurrence	Model With Occurrence / All Models
AssCyc	50	92.59%
ImpAbs	39	72.22%
RelRig	37	68.52%
RepRel	31	57.41%
MultDep	28	51.85%
UndefFormal	27	50.00%
BinOver	26	48.15%
RelSpec	26	48.15%
RelComp	24	44.44%
HomoFunc	20	37.04%
FreeRole	18	33.33%
RelOver	12	22.22%
DeclInt	10	18.52%
UndefPhase	8	14.81%
HetColl	7	12.96%
WholeOver	6	11.11%
GSRig	5	9.26%
DepPhase	5	9.26%
MixIden	5	9.26%
PartOver	1	1.85%
MixRig	1	1.85%

The percentage of models with at least one occurrence can also mislead the results' interpretation. The frequency that modelers use a particular element type directly affects the frequency of anti-patterns defined in terms of that element. For example, the RelRig anti-pattern, defined as relator connected to rigid types, can only occur in a model that has relators and rigid types.

To remove from the frequency evaluation the impact of a particular element type usage, we compare the number of models with at least one occurrence of an anti-pattern with the models with at least one element that can characterize the anti-pattern. In a way, this comparison shows the rate of “dangerous” usage of a particular element type. Table 58 presents the results by additionally identifying the relevant element type, the number of models that contain at least one instance of the respective model element, and new percentage. Note that, for some anti-pattern, like AssCyc, no change is identified, since the required element type occurs in all models.

Table 58. Anti-Pattern frequency on models with required elements.

Anti-Pattern	Relevant Element Type	Models with Occurrence	Models With Element	Models With Occurrence / Models With Element Type
UndefFormal	Formal	27	29	93.10%
AssCyc	Association	50	54	92.59%
RelRig	Relator	37	48	77.08%
ImpAbs	Association	39	54	72.22%
RepRel	Relator	31	48	64.58%
MultDep	Mediation	28	48	58.33%
HomoFunc	Meronymic	20	41	48.78%
BinOver	Association	26	54	48.15%
RelSpec	Association	26	54	48.15%
RelComp	Association	24	54	44.44%
UndefPhase	Phase	8	21	38.10%
FreeRole	Role	18	49	36.73%
RelOver	Relator	12	48	25.00%
DepPhase	Phase	5	21	23.81%
Declnt	Sortal	10	54	18.52%
HetColl	Meronymic	7	41	17.07%
WholeOver	Meronymic	6	41	14.63%
MixRig	Mixin	1	7	14.29%
GSRig	Gen. Set	5	37	13.51%
MixIden	Non Sortal	5	37	13.51%
PartOver	Meronymic	1	41	2.44%

Surprisingly, from this perspective, AssCyc loses the position of most frequent anti-pattern. We encounter at least one UndefFormal occurrence in 27 of the 29 models that have formal relations, resulting in an occurrence rate of 93.10%. HomoFunc’s frequency also significantly rises, from 37% to 48%. If we aggregate the results once more, we see three anti-patterns in more than 75% of the relevant models, another three from 50-75%, seven from 25-50% and eight in 25% or less.

The last variable we use to evaluate anti-pattern frequency is the ratio between the number of elements and the number of anti-pattern occurrences. For example, if ten relators characterize two occurrences of RelRig, the ratio is of five relators per RelRig occurrence. This measure will help us understand the rate in which users produce anti-patterns in their ontologies, taking into account the elements that cause the problem and ignoring the size of the model and the frequency of use of a particular element.

For each anti-pattern, Table 59 presents:

- the number of occurrences identified in all models (#Occ.),
- the type of element most relevant to the anti-pattern,
- the number of the element type encountered in all models (#Element),
- the rate between the number of elements and the number of occurrence considering all models (#Element/#Occ.) and lastly,
- the average rate of elements per occurrences considering only the models that have at least one occurrence of the anti-pattern.

Table 59. Anti-pattern appearance rate regarding model elements.

Anti-Pattern	#Occ.	Element Type	#Element	#Element/#Occ.	#Element/#Occ. (Model Average)
UndefFormal	293	Formal	373	1.27	1.07
AssCyc	1809	Association	4017	2.22	7.23
RepRel	319	Relator	1204	3.77	2.94
RelRig	282	Relator	1204	4.27	4.11
RelSpec	817	Association	4017	4.92	9.90
HomoFunc	142	Meronymic	735	5.18	4.46
ImpAbs	758	Association	4017	5.30	6.89
RelComp	739	Association	4017	5.44	6.91
RelOver	149	Relator	1204	8.08	7.27
MixRig	6	Mixin	53	8.83	12.33
HetColl	60	Meronymic	735	12.25	6.47
FreeRole	119	Role	1930	16.22	9.91
BinOver	224	Association	4017	17.93	14.32
MultDep	105	Mediation	1908	18.17	31.76
UndefPhase	11	Phase	209	19.00	9.56
WholeOver	27	Meronymic	735	27.22	10.48
DepPhase	7	Phase	209	29.86	13.90
MixIden	16	Non Sortal	622	38.88	8.74
PartOver	13	Meronymic	735	56.54	24.77
GSRig	16	Gen. Set	1080	67.50	27.60
Declnt	92	Class	6744	73.30	44.43

The element/anti-pattern rate is the most precise characterization of anti-pattern frequency we provide because it ignores the overall size of the model and the usage frequency of the required elements. This rate provides an estimation of the number of elements required for a modeler to produce an anti-pattern. Again, *.UndefFormal* appears at the top: for every 12 associations, 10 characterize an occurrence. This result is surprising because it shows that domain comparative formal relations are rarely used or that modelers do not know, want or need to specify them properly (providing the required qualities and derivation rule).

We can also see that 10 anti-patterns require less than 10 usages of their respective element type to occur, five require from 10-20 usages, and the remainder six requires more than 20.

To conclude whether or not our catalogue is of recurrent modeling decisions, one must define what “recurrent” means. Does it account for how many overall occurrences we identified? How many different modelers made that decision? Alternatively, even how many elements modelers use in average to generate an anti-pattern occurrence? Despite what one might consider the most relevant characteristic, this study shows that the proposed catalogue is composed of recurrent modeling decisions, even though some might be more frequent than others might.

7.2 ACCURACY EVALUATION: THE MGIC ONTOLOGY

In this second study, we focus on anti-pattern **accuracy**, instead of frequency. The goal is to measure two things: the probability of an anti-pattern occurrence to characterize a domain misrepresentation and how often users select the provided refactoring plans to solve an occurrence that in fact characterizes a mistake. Through the analysis of these two variables, we assess the usability of an anti-pattern.

It is not possible to conduct a study of this nature in a fully automatic way, as we did in the first study. In fact judging an anti-pattern occurrence a mistake is a matter of domain knowledge and design decisions made by the modeler. With that in mind, we chose to conduct the anti-pattern accuracy evaluation as a case study using the MGIC ontology. Multiple factors led to this decision:

- the ontology is the biggest in the repository;
- it contains occurrences of all anti-pattern types;
- ten modelers participated in its development, throughout three years;
- it is the product of an industrial project with the Brazilian government; and
- most importantly, the modelers accepted to participate in the analysis because it was their interest to validate their ontology.

7.2.1 Methods and Tools

Eight modelers participated in this empirical study. We assigned sub-ontologies to each of them, taking into account their knowledge about the domain and its complexity, represented by the number of classes and relations used to formalize it. To guarantee that the modelers would have enough knowledge to analyze the anti-pattern occurrences, we mostly assigned subdomains that he/she participated in the development. We also encouraged modelers to interact with each other during the case study.

The modelers developed the MGIC ontology using Sparx Enterprise Architect¹⁴, a UML-based modeling tool. In order for modelers to use the anti-pattern detection tool, they had to export the whole model in the XMI format and then import into OLED the subdomains they were assigned. Modelers conducted the anti-pattern detection exclusively through the tool. Furthermore, modelers analyzed the anti-pattern using the wizards we implemented (for more details, see Chapter 8).

We intentionally did not provide participants with any anti-pattern training. Not regarding their structure, justification or predicted solutions. We also did not specify any order in which the participants should analyze the anti-patterns. We made these decisions because our goal, since the beginning of this research, was to develop a tool that did not require formal training. In fact, that is one of the reasons that we implemented a wizard to guide modelers throughout the analysis of each anti-pattern.

¹⁴ <http://www.sparxsystems.com.au/>

By not providing any training, this study also manages to provide us feedback regarding how user friendly our proposal is.

We highlight that, throughout the development of this case study, the anti-pattern management tool was improved. It was an interactive process: the participants used it and if whenever they identified bugs or improvements opportunities, they contacted us so we could discuss and improve the anti-pattern support. We were available to answer all sorts of question to the participants, either regarding anti-pattern definition, refactoring plans or even ontological notions required to understand the anti-patterns.

Therefore, what we asked the participant to do was to manually analyze each anti-pattern occurrence identified within his assigned subdomain and register his conclusion according to the following template:

- **Anti-Pattern Type:** an acronym to identify the type of anti-pattern
- **Description:** a textual description automatically generated by the anti-pattern tool. It identifies the classes and associations relevant for understanding the anti-pattern. Moreover, it is useful to reproduce each occurrence.
- **Decision:** a binary field that captures the ultimate decision regarding an anti-pattern occurrence. The field can be set as “Error” or “Correct”. Participants used the former if the occurrence analysis lead to some modifications in the model, predicted by our anti-patterns or not. We intentionally did not provide a “Don’t Know / Don’t Understand” option because, in those cases, we instructed the modelers to interact with us, if the doubt regarded anti-pattern definition, or to interact with each other, to solve a decision question.
- **Action:** describes the action participants adopted to refactor the model – we instructed participants to input information in this field only if they identify an error.
- **Predicted:** three values can be assigned to this field for this field: “Yes”, “No” and “Partially”. It describes whether anti-pattern was completely able, partially able or unable to predict the refactoring actions.
- **Comment:** a field that participants could freely fill (e.g. doubts, intuitions, observations, and so on).

Due to the size of the MGIC ontology (3800 classes and 1800 associations), we expected the number of identified anti-patterns to be also very large. This case study did not intend to analyze every single anti-pattern occurrence, but as many as possible. We partially analyzed the anti-patterns named AssCyc, RelComp and ImpAbs, since we encountered more than 400 occurrences of each.

7.2.2 The MGIC Ontology

Before we present the study results, we briefly describe the MGIC ontology, providing an overview of its development context, the domain it formalizes and its structural information (number of classes, relations, stereotypes, etc.)

The project entitled *Modelo de Gestão da Informação e Conhecimento*¹⁵ (MGIC) is a product of a partnership between a Brazilian regulatory agency named *Agência Nacional de Transportes Terrestres*¹⁶ (ANTT) and Brazilian federal universities of Espírito Santo, Fluminense and Rio de Janeiro (UFES, UFF and UFRJ respectively). It was conceived to improve the way ANTT manages information and knowledge, by means of integration and consolidation (BASTOS et al., 2011). The adopted methodology proposed the creation of five types of models: information flow, business requirements and assets, knowledge and competence, and an ontology-based reference conceptual model.

The ontology's main role was to provide structure and semantics to the information handled by the agency, to serve as reference model to allow semantic interoperability between the systems controlled it maintains (BASTOS et al., 2011). The MGIC ontology also intended to serve as a guide to the agency's databases triplication and posterior publication. Information transparency and publication is a law-imposed obligation for all Brazilian governmental organizations since the sanction of the law entitled *Lei de Acesso à Informação*¹⁷ (BRASIL, [s.d.]).

¹⁵ In English: Knowledge and Information Management Model

¹⁶ In English: National Ground Transportation Agency

¹⁷ In English: Information Access Law

The design of the MGIC ontology took 3 years. Throughout that time, 10 modelers were involved, who collaborated with close to 40 domain experts in order to define the scope and capture the conceptualization shared within the agency. A team of ontologists visited the 11 main departments of the agency. In each department, they interviewed experts appointed by the departments' management.

The ontology describes the domains relevant for ground transportation regulation. The following list presents the most relevant ones:

- *Cargo transportation*: includes definitions related to cargo transportation by truck, train, pipelines or multimodal (a combination of different transport types). It describes the differences between interstate and international cargo transports, and transportation of hazardous products.
- *Passenger transportation*: describes concepts related to interstate and international regular and eventual passenger transportation on both highways and railroads.
- *Infrastructure concession*: describes the process of concession and controlling of highway and railroads infrastructure to private companies.
- *Legislation*: includes concepts regarding the legal process for regulating the transport segment.

From a structural perspective, the MGIC Ontology is massive, particularly if compared to other conceptual models. It defines 3800 classes, 1918 associations, 3616 generalizations, 698 generalization sets, 71 data types, 865 attributes and 149 constraints, all distributed in 291 packages. To the extent of our knowledge, this is the biggest OntoUML model ever created!

The ontology also has occurrences of every single language construct defined in OntoUML: all eleven class' stereotypes and all nine association's stereotypes. Figure 54 details the number of occurrences for each class stereotype. Notice that Role is the most frequent stereotype: 1066 occurrences or 28.1% of all classes.

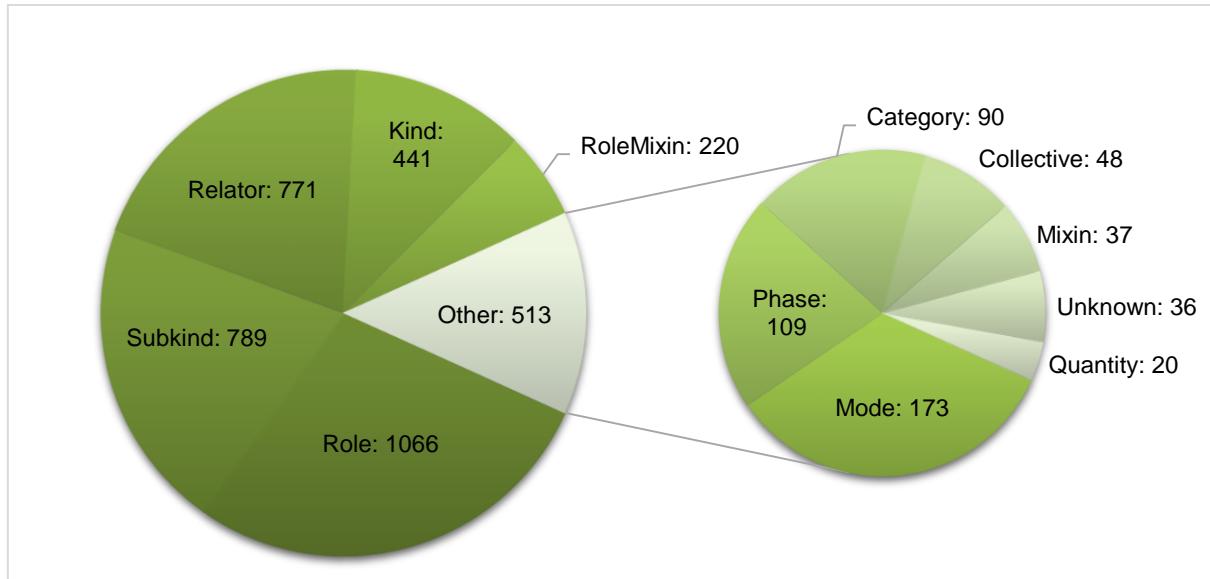


Figure 54. Distribution of the class' stereotypes within the MGIC Ontology.

Figure 55 depicts the associations stereotype distribution in the MGIC ontology. The number of mediations is significantly higher than the others, 1103 or 57.5% of all associations in the model. We expected this high number of mediations, since they are required to characterize roles.

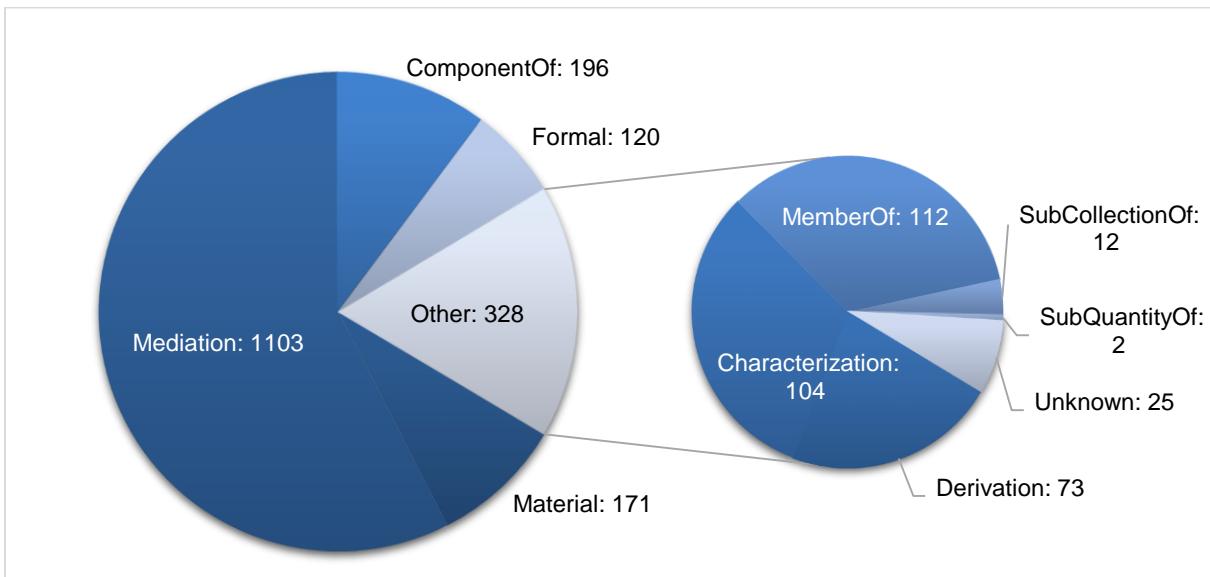


Figure 55. Distribution of association's stereotypes within the MGIC Ontology.

7.2.3 General Results

The modelers analyzed together 1475 anti-pattern occurrences in the MGIC ontology. We summarize the results in Table 60. The column identified as “#Occ.”, stands for the number of analyzed occurrences of a given anti-pattern type, whilst the one labeled as “#Error”, refers to the number of occurrences considered as modeling errors by the participants. The columns “#Pred.”, “#Partial” and “#Custom” stand for the sum of occurrences the participants fixed using: exclusively refactoring plans, some refactoring plans and some custom solutions and exclusively custom solutions, respectively.

We measure the accuracy of anti-pattern by dividing the number of times it characterizes a mistake by the number of times it occurs. This measure provides the probability of a given occurrence to characterize a mistake. If we sum all occurrences of all types, we have that in 53.8% of the cases an anti-pattern characterizes a mistake. Roughly, for every two occurrences, one is an error and one is a “false alarm”.

Individually analyzing the results, we notice that some anti-patterns characterized mistakes in every single occurrence, like GSRig, Mixlden and MixRig. Conversely, ImpAbs’s problem rate was the lowest, only 8.8% of the time. Furthermore, if we classify anti-pattern by accuracy range, we have: five anti-patterns with a problem rate greater than 75%, another nine in 50-75%, four resulted in errors in 25-50% of the time, and only two in less than 25%. These numbers are a strong indication that the structures identified by the anti-patterns are indeed error-prone.

Anti-pattern accuracy also refers to the capacity of predicting appropriate refactoring solutions. We measure that by dividing the number of occurrences in which modelers exclusively adopted standard solutions by the number of occurrences that they considered as mistakes. In this study, the modelers exclusively adopted pre-defined solutions 692 times in the 794 occurrences considered as errors. This represents a percentage of 87.15% of the time, in comparison to 2.39% of partial solutions and 10.45% of exclusively custom ones.

Taking each anti-pattern at a time, we notice two that were able to provide appropriately solutions a hundred percent of the time: DepPhase and HetColl. RelSpec

and RelRig almost hit the same level, with 97.1% and 98.1% respectively. On the other end, conversely, we had not so promising results for RelComp, PartOver and ImpAbs, with a predictability of 35.3%, 33.3% and 27.3%, respectively.

Table 60. Summary of anti-pattern accuracy results.

Anti-Pattern	#Occ.	#Error	#Error / #Occ.	#Pred.	#Pred. / #Error	#Partial	#Partial / #Error	#Custom	#Custom / #Error
DepPhase	4	2	50.0%	2	100.0%	0	0.0%	0	0.0%
HetColl	52	11	21.2%	11	100.0%	0	0.0%	0	0.0%
RelRig	161	107	66.5%	105	98.1%	1	0.9%	1	0.9%
RelSpec	315	279	88.6%	271	97.1%	1	0.4%	7	2.5%
GSRig	16	16	100.0%	15	93.8%	0	0.0%	1	6.3%
WholeOver	17	16	94.1%	15	93.8%	1	6.3%	0	0.0%
UndefFormal	96	43	44.8%	40	93.0%	0	0.0%	3	7.0%
RepRel	221	57	25.8%	48	84.2%	4	7.0%	5	8.8%
FreeRole	39	23	59.0%	19	82.6%	2	8.7%	2	8.7%
Declnt	55	17	30.9%	14	82.4%	0	0.0%	3	17.6%
RelOver	124	70	56.5%	54	77.1%	1	1.4%	15	21.4%
MixIden	13	13	100.0%	10	76.9%	2	15.4%	1	7.7%
BinOver	74	31	41.9%	23	74.2%	0	0.0%	8	25.8%
HomoFunc	61	33	54.1%	24	72.7%	0	0.0%	9	27.3%
AssCyc	20	14	70.0%	10	71.4%	0	0.0%	4	28.6%
MultDep	41	23	56.1%	16	69.6%	6	26.1%	1	4.3%
MixRig	6	6	100.0%	4	66.7%	0	0.0%	2	33.3%
UndefPhase	3	2	66.7%	1	50.0%	1	50.0%	0	0.0%
RelComp	28	17	60.7%	6	35.3%	0	0.0%	11	64.7%
PartOver	4	3	75.0%	1	33.3%	0	0.0%	2	66.7%
ImpAbs	125	11	8.8%	3	27.3%	0	0.0%	8	72.7%
Total	1475	794	53.8%	692	87.15%	19	2.39%	83	10.45%

7.2.4 Individual Results

This study also generated results that contribute in understanding anti-patterns individually. For one, we gathered the frequency that users chose each refactoring plan. Furthermore, we even managed to capture, for some anti-patterns, a recurrent argumentation to justify occurrences not being errors.

In the following sections, we discuss relevant individual results for the BinOver, FreeRole, HomoFunc and UndefFormal anti-patterns. For the complete detailed results, please refer to Appendix B.

7.2.4.1 BinOver

BinOver, the anti-pattern characterized by an association between overlapping types, occurred 74 times. From those, 31 were actual mistakes (41.9%). Counting only the mistakes, the modelers adopted one of our suggested solutions 23 times (74.2%). The option to enforce one or more binary properties was selected 16 times (70%), while the alternative to enforce disjointness between the related types was selected seven times (30%). The modelers did not opt to change the stereotype of the relation for any occurrence.

If we inspect the types of enforced binary properties, we see that anti-reflexivity and anti-symmetry were the most common ones, being set 15 and 14 times respectively. The acyclic constraint follows, requested 9 times. The need to specify transitive, reflexive or symmetric relations was only encountered one, two and one time respectively. This is an indication that the need of binary properties is there and we might try to proactively incentive modelers to specify such constraints.

We also managed to identify the reasons that lead the modelers not to consider a BinOver occurrence to be a mistake. From the 43 correct cases, in 19 times (44.2%) they considered the relation under analysis as derived and the binary properties were consequence of the embedded derivation. In another 11 times (25.6%), the desired binary properties came from the stereotype choice. Furthermore, in seven cases (16.3%), the participants fixed the problem unintentionally, through a solution of a previous anti-pattern. Lastly, for the remainder 6 cases, participants did not provide further justification.

7.2.4.2 FreeRole

The FreeRole anti-pattern is characterized by a role that is directly connected a mediation and has one or more direct role subtypes which are not defined by any extra relational dependency. Because it aggregates many free roles in the same occurrence, the overall number of identified occurrences is not as high as other anti-patterns: only 39 occurred in the MGIC ontology. From those, 23 characterized modeling problems (59%).

Although we propose four different refactoring options for the FreeRole anti-pattern, the participants of the case study selected only three. From the 23 considered as mistakes, 9 times the solution was to define the role as derived by intersection. Another nine followed the sub-role pattern and in 8 times, the role of role pattern. The participants did not choose the material-defined role pattern for any occurrence. Note that more than one role specialization pattern could be selected per occurrence.

Now, considering the 16 times in which modelers considered FreeRole identifications as false alarms, we manage two separate four different rationales. First, 12 occurrences characterized role specializations defined by the RoleMixin anti-pattern. In fact, this presents a flaw in the specification of the anti-pattern structure, which should never consider such cases. The other frequent justification was that meronymic relations defined the role specializations. Currently, OntoUML does not support this modeling pattern, since mediations, not meronomies, are the relations able to define roles.

7.2.4.3 *HomoFunc*

HomoFunc is an anti-pattern characterized by a whole composed by a unique type of part. It occurred 61 times in the MGIC ontology, which the participants considered as mistakes 33 times (or 54.1% of the time). Its predictability is high: in 24 of the 33 erroneous occurrences, participants adopted a predicted solution. In half of them, it implied the creation of additional functional parts, whilst in the other half the transformation to a membership relation.

What we did not foresee was to provide modelers the option to create an additional part type in an ancestor of the type that plays the Whole *HomoFunc*. In fact, that was the solution in eight cases. Another solution that was only required once, but which theoretically makes a lot of sense, is to change the stereotype of the single componentOf relation into a non-meronymic one, like a material relation, or even formal one.

Looking at the justifications for not considering an occurrence a mistake, we identified a quite diversified set. Most frequently, for 19 occurrences, modelers claimed that even

if they wanted to change the functional part-hood to a membership, they could not. The reason was that the type identified as the whole was actually a functional part of another whole, which in turn had different part types. Figure 56 depicts an example of this dilemma: a simplified fragment of the MGIC ontology about the legislation domain. The main concept in the diagram is Normative Act, the general classification used for all types of legal documents publishable by the entities of the Brazilian government, like a law, a presidential decree or a resolution. These acts are composed amongst other things by an Epigraph, the top part of the act that qualifies the type and situates it in time (e.g. Law nº 1234 of May 3rd, 2014), and Articles, the basic division unit of an act. What characterizes a HomoFunc occurrence, however, is a part of the article, named Item, which only has letters as parts. If one analyzes solely the composition between items and letters, one might conclude that it is a relation between a collection, the Item, and its members, the letters. That conclusion is invalidated by noticing that items are functional parts of an article, which is a functional part of a normative act.

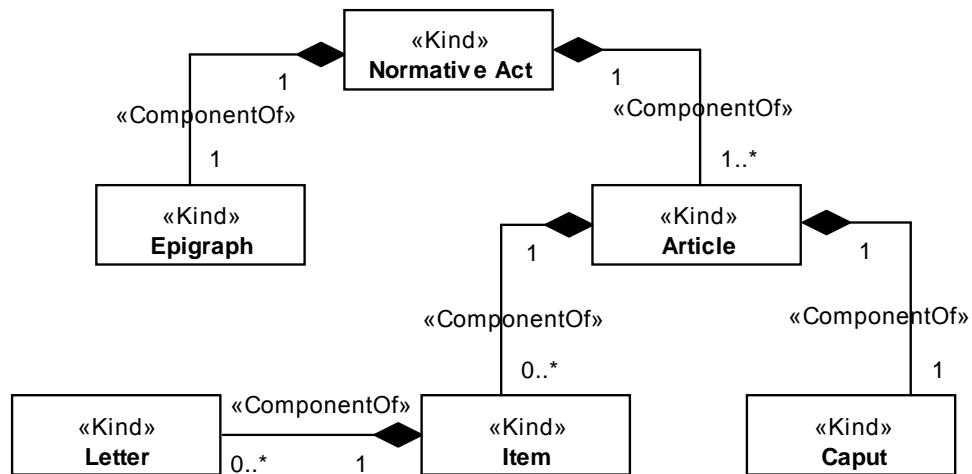


Figure 56. Simplified and translated excerpt of the MGIC ontology which about the regulation domain

7.2.4.4 *UndefFormal*

The UndefFormal anti-pattern occurs due to an improper use of the formal stereotype. We argue that, instead of using it to specify formal domain comparative formal relations, like “being older than”, or “being heavier than”, modelers use it as an escape

route when they are not able to fit any other stereotype or do not have the necessary knowledge to do so.

In the 96 identified occurrences, participants considered it a modeling error in 43 cases. From those, 38 times the participants decided to change stereotype of the relation, corroborating assumption of improper use of the formal stereotype. In fact, in 26 times, they concluded that the relation should be a componentOf. Furthermore, in other 11 cases, they changed the relation's stereotype to material.

As usual, through the analysis of the occurrences that the participants considered correct, we identified that in many times, modelers use the formal stereotype to capture a type of relation that OntoUML does not cover. In 19 times, the relation was defined connected at least one higher-order universal (a still unsupported class type whose instances are other classes, very similar to UML's idea of powertype (OMG, 2011b)). This presents a demand for expand OntoUML to cope with both this type of relation and this type of class.

Furthermore, the investigation led to the identification of a particular design problem that modelers recurrently solved with the formal stereotype: how to model the relation between a concrete material thing (e.g. person, ball) and the photograph someone took of it. It is not a meronymic relation because it does not account for weak supplementation – a photo can depict only one person. It is not domain comparative because it is not reducible to a comparison between qualities. Furthermore, it is not a material relation because there is no ongoing process that provides relational characteristics for the related elements. All we have is that an event occurred: one took the picture, and while it exists, it will always depict the very same things. There seems to be a sort of historical dependency, which no OntoUML stereotype currently covers. Concrete examples identified in the MGIC ontology are the relation between buildings and blueprints and the relation between bus passenger lines and exploration projects.

Lastly, we identified another controversial domain specific problem, which regards specifying that someone owns something. No consensus was reached amongst the participants if whether it is a material relation or a formal one (in the generic sense). We see here an open question on how to proper model this domain specific pattern in OntoUML.

7.3 CONCLUSION

To provide a conclusion on anti-pattern evaluation, we cross frequency and accuracy information on Table 61. For the frequency column, we consider the percentage of models an anti-pattern was encountered per models in which they could occur (had at least one instance of the anti-pattern's main element type). For the problem rate and predictability columns, we considered the results of the MGIC case study. In the former the rate between occurrences that characterized errors per total number of occurrences, and in the latter, the percentage of erroneous occurrences that one could fix using exclusively pre-defined solutions. Furthermore, instead of the actual percentages, we adopted a discrete scale to classify the values, containing specified as follows: Very High (80-100%), High (60-80%), Medium (40-60%), Low (20-40%) and Very Low (0-20%).

The higher all these three values are for an anti-pattern, the more useful it is for ontology validation. Anti-patterns that always occur, with a high possibility of characterizing a mistake and being able to predict most of the refactoring necessities are more likely to be useful during ontology validation. Examples of such anti-patterns are AssCyc, RelRig and RelSpec.

“Bad” anti-patterns, on the other hand, are not the scarcely identified ones, but the ones that we frequently find but rarely are the source of domain misrepresentations. In fact, they require a lot of effort to analyze and little gain in ontology quality. ImpAbs is an example of anti-pattern that needs refinement. Our analysis of the phenomenon is that ImpAbs’ generic structure, which encompasses all association and class stereotypes, increases the number of occurrences. In this case, a refinement is in order. Furthermore, ImpAbs’ low predictability corroborates the need to improve not only its structural definition but also the associated refactoring plans.

We argue that the combination of the frequency and accuracy anti-pattern studies where successful in the evaluation of anti-pattern usability. Not only that, but it improved our confidence that they can be an important tool in ontology validation.

An unexpected consequence of manually inspecting a great number of anti-pattern occurrences is that it provided feedback for anti-pattern refinement opportunities.

Furthermore, it provided insights on how OntoUML is used and what else modelers demand from it, like meronymic-defined roles, and constitution relations.

Table 61. Summary of the evaluation results from both studies.

Anti-Pattern	Frequency	Problem Rate	Predictability
AssCyc	Very High	High	High
BinOver	Medium	Medium	High
Declnt	Very Low	Low	Very High
DepPhase	Low	Medium	Very High
FreeRole	Low	Medium	Very High
GSRig	Very Low	Very High	Very High
HetColl	Very Low	Low	Very High
HomoFunc	Medium	Medium	High
ImpAbs	High	Very Low	Low
Mixlden	Very Low	Very High	High
MixRig	Very Low	Very High	High
MultDep	Medium	Medium	High
PartOver	Very Low	High	Low
RelComp	Medium	High	Low
RelOver	Low	Medium	High
RelRig	High	High	Very High
RelSpec	Medium	Very High	Very High
RepRel	High	Low	Very High
UndefFormal	Very High	Medium	Very High
UndefPhase	Low	High	Medium
WholeOver	Very Low	Very High	Very High

8 TOOL SUPPORT

In this chapter, we describe the implementation strategies for the proposed validation techniques. Particularly, we discuss the implementation of the refactored transformation from OntoUML to Alloy in Section 8.4 and the support for managing anti-patterns in OntoUML models in Section 8.5.

8.1 THE ECLIPSE MODELING FRAMEWORK

The Eclipse Modeling Framework (EMF) is a modeling framework that exploits the facilities provided by Eclipse (STEINBERG et al., 2008). It bridges the worlds of modeling and programming, by integrating, through automatic transformations, three important technologies: UML, Java and the *eXtension Markup Language* (XML). EMF brings all these technologies together by defining a common core of concepts between them, formalized in the ECore modeling language, the center of the EMF approach.

EMF is a framework that moves towards model-driven architecture (MDA), because, although it considers modeling and programming the same thing, the level of abstraction captured by the modeling language is significantly low. EMF's most relevant feature for this work is the capability of generating Java code from an ECore model. With that, we are able to implement a series of model manipulations, like dynamic instance construction, model-to-model transformations, syntactic and semantic validation, amongst others.

EMF's modeling language, ECore, is very useful for defining the abstract syntax of a language. In fact, the EMF environment provides many features for implementing textual and graphical editors based on ECore-defined languages. For more details about EMF, its applications and tool support, please refer to (STEINBERG et al., 2008).

8.2 THE ONTOUML META-MODEL

All software components developed in this work use the version 1.1.0 of the Reference OntoUML (RefOntoUML) metamodel¹⁸. The current version of the metamodel corresponds to Carraretto's initial proposal (2010), updated with Albuquerque's proposal for quality representation (ALBUQUERQUE; GUIZZARDI, 2013) and some other minor improvements.

Implemented in ECore and enriched with OCL constraints, RefOntoUML is quite complex. A complete description is provided in (CARRARETTO, 2010). The author details all attributes and operations defined within the meta-model, alongside with its syntactical constraints.

8.3 OLED – ONTOUML LIGHTWEIGHT EDITOR

The OntoUML Lightweight Editor (OLED)¹⁹ is an open-source front-end tool for building and manipulating OntoUML models. Distributed as a runnable .jar, OLED is compatible with the latest versions of the most common operational systems: Mac OS, Windows and Linux.

OLED is more than just a CASE tool for OntoUML. OLED is a modeling environment that aggregates the technological results of OntoUML-related researches. It provides a number of additional functionalities that support users throughout model development. Firstly, it provides automatic syntactical verification, obtained from the implementation of the latest RefOntoUML version. It also provides support for the specification, verification and validation of OCL constraints, resultant from the research described in (GUERSON; ALMEIDA; GUIZZARDI, 2014). Moreover, the tool provides alternative transformations from OntoUML to OWL (BARCELOS et al., 2013; ZAMBORLINI; GUIZZARDI, 2010) in addition to model verbalizations in either SBVR and natural language. Now, due to the results obtained from this research, it also

¹⁸ Download available at: <https://code.google.com/p/ontouml-lightweight-editor/>

¹⁹ OLED's latest compiled version and source code are available at: <https://code.google.com/p/ontouml-lightweight-editor/>

supports model validation through visual simulation and anti-pattern management. In the remainder of this chapter, we discuss in more detail these last two components.

During the writing of this thesis, OLED features version 0.9.34. Figure 57 depicts a screenshot taken of OLED running on Windows 8.1. The left compartment, entitled “Toolbox”, provides OntoUML’s basic modeling constructs. The right compartment is the “Project Explorer”, a tree-like representation of the model. On the center, the “Editor” compartment can open several tabs containing either diagrams or OCL documents. Finally, on the bottom, the “Footer” compartment provides feedback for operations performed within the tool, like syntactical checking.

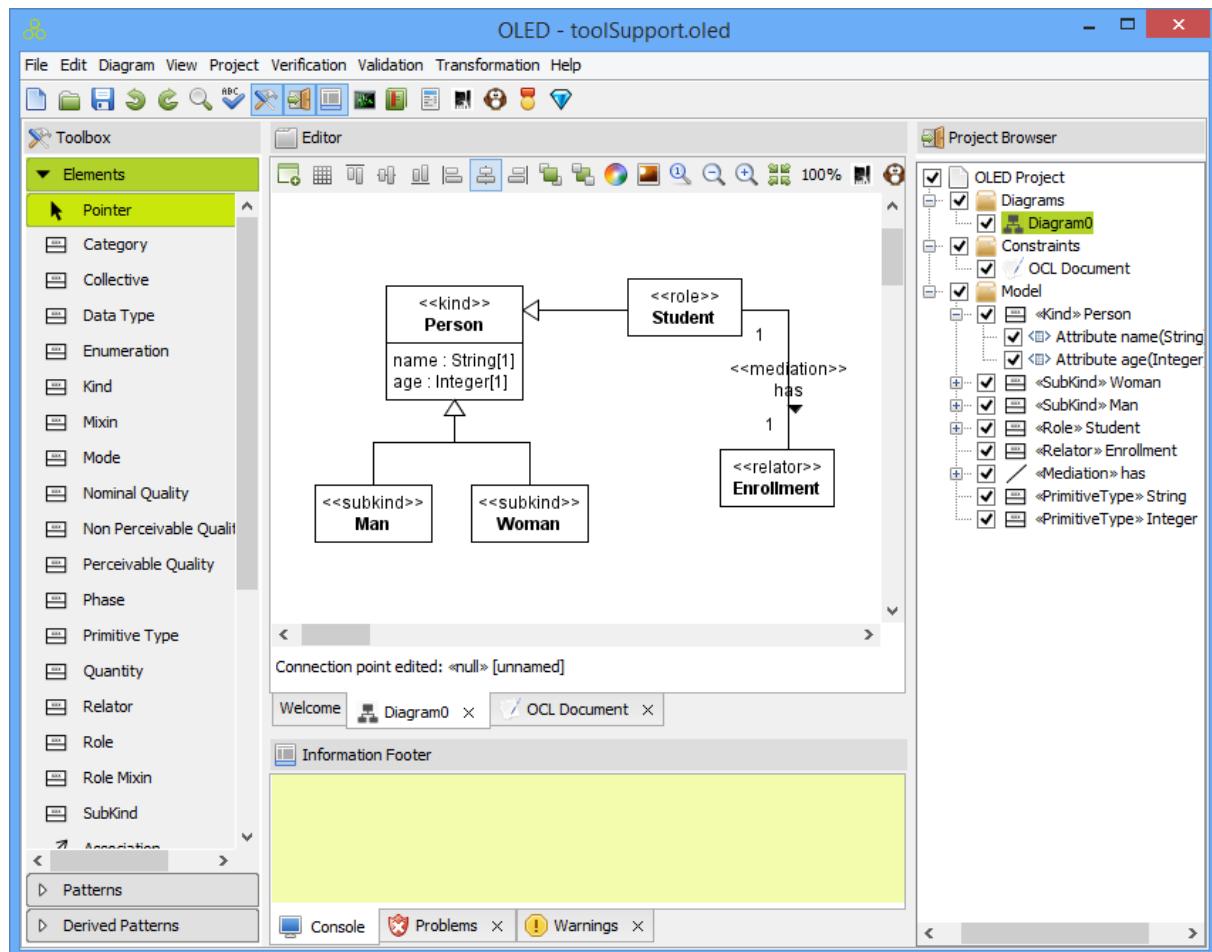


Figure 57. OLEDv0.9.34 screenshot running on Windows 8.1.

8.4 THE SIMULATION COMPONENT

This section elaborates on the technological support required for simulating OntoUML models: the implementation of the refactored OntoUML2Alloy transformation and the user interface in OLED.

We implemented our refactored transformation from OntoUML to Alloy in two complementary steps. First, we perform a model-to-model transformation: from RefOntoUML to the Alloy metamodel presented in the following subsection. In the following, we conduct a model to text transformation: a serialization of the Alloy metamodel instance in the textual concrete syntax.

We distribute the transformation within OLED and implement a series of features to improve usability.

8.4.1 The Alloy Metamodel

The metamodel presented in this section is the ECore representation of the Alloy 4.0 syntax, defined in (JACKSON, 2012) and represented in Extended Backus Normal Form (EBNF).

The Alloy metamodel plays a role of “middle man” in our implementation of the transformation of OntoUML models into Alloy specifications. It separates the mapping between concepts of the abstract syntax (the actual transformation), from concerns related to the concrete syntax, like ordering, indentation, etc.

We present the metamodel incrementally, dividing it in 4 diagrams. We present them individually, alongside their respective EBNF specification. Four diagrams compose the metamodel: Module Composition, Paragraph Composition, Command and Expression. We discuss the meaning of each Alloy construct in Annex A.

8.4.1.1 Module Composition

The diagram presented in Figure 58 depicts the possible composition of an Alloy specification, based on the EBNF definition of Listing 20. The *AlloyModule* class corresponds to the container of all other classes (like a Package for UML or OntoUML models). It can optionally contain module importations, signature parameters and paragraphs.

Listing 20. Alloy's module composition in EBNF.

```

alloyModule ::= [moduleDecl] import* paragraph*
moduleDecl ::= module qualName [[name,+]]
import ::= open qualName [[qualName,+]] [as name]
paragraph ::= sigDecl | factDecl | predDecl | funDecl | assertDecl | cmdDecl

```

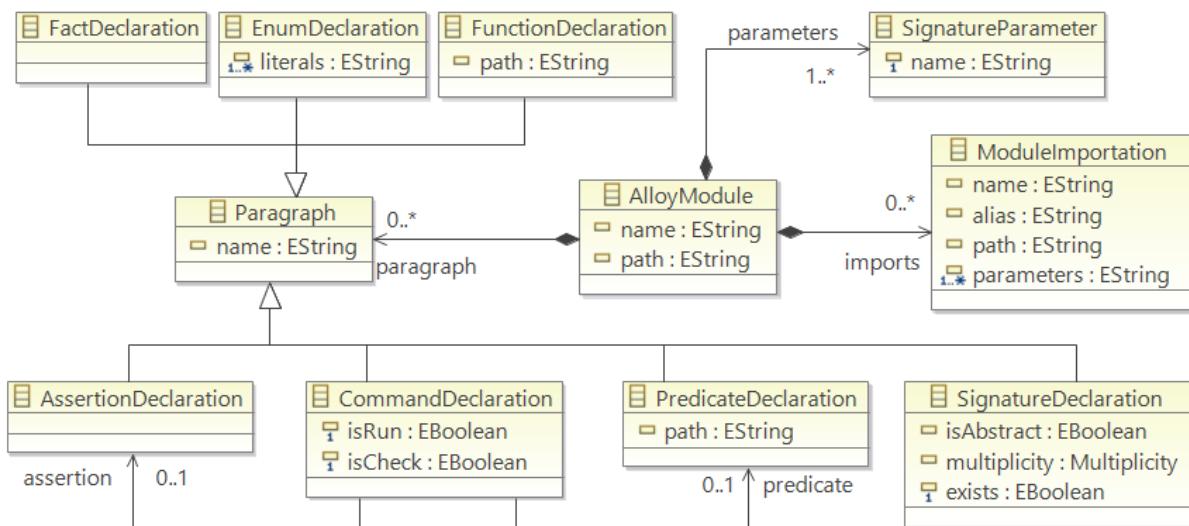


Figure 58. Alloy meta-model fragment: module composition.

8.4.1.2 Paragraph Properties

The diagram on Figure 59 depicts the attributes and relational properties of the different types of paragraph in Alloy, as defined in Listing 21: signature, fact, predicate, function and assertion declarations. All these types of paragraphs (an artificial concept we created to improve the structure of the abstract syntax) are composed by blocks, which in turn are composed by expressions.

Listing 21. Alloy's paragraph properties in EBNF.

```

sigDecl ::= [abstract] [mult] sig name, + [sigExt] { decl,* } [block]
sigExt ::= extends qualName | in qualName [+ qualName]*
qualName ::= [this/] (name /)* name
mult ::= lone | some | one
decl ::= [disj] name, + : [disj] expr
factDecl ::= fact [name] block
predDecl ::= pred [qualName .] name [paraDecls] block
funDecl ::= fun [qualName .] name [paraDecls] : expr { expr }
paraDecls ::= ( decl,* ) | [ decl,* ]
assertDecl ::= assert [name] block
block ::= { expr* }

```

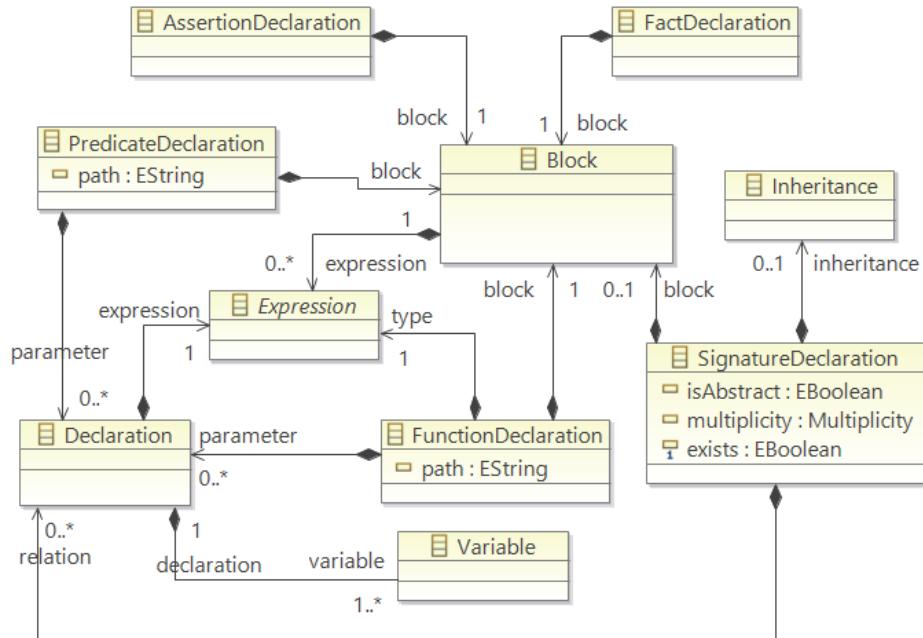


Figure 59. Alloy meta-model fragment: paragraph composition.

8.4.1.3 Command Paragraph

Figure 60 presents a diagram defining the last type of paragraph in Alloy: the run and check commands. Like the other paragraphs, it may also contain blocks. Moreover, it has optional references to predicate and assertion declarations (the executable paragraphs). Listing 22 provides the official definition of command declarations in EBNF.

Listing 22. Alloy's command declaration in EBNF.

```

cmdDecl ::= [name :] [run | check] [qualName | block] [scope]
scope ::= for number [but typescope,+] | for typescope,+
typescope ::= [exactly] number qualName
qualName ::= [this/] (name /)* name
block ::= { expr* }

```

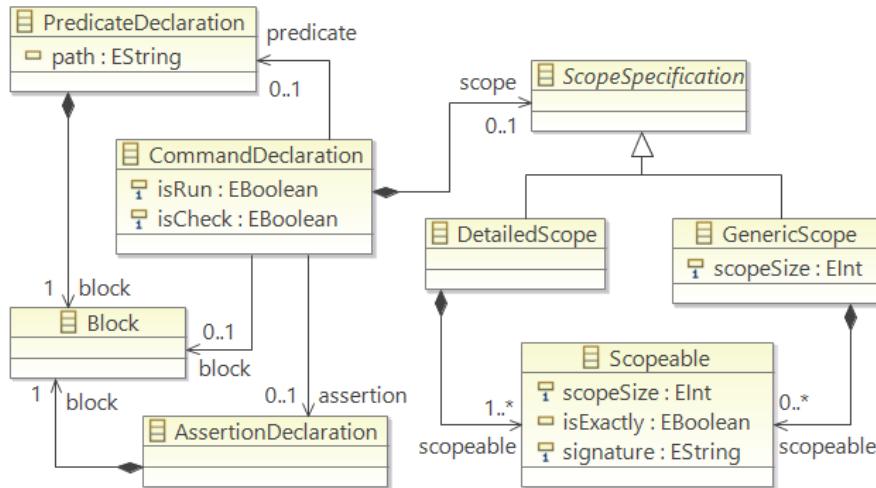


Figure 60. Alloy meta-model fragment: command paragraph.

8.4.1.4 Expression

The last part of the Alloy metamodel defines the possible way to define expressions in Alloy. Figure 61 depicts how expressions are recursively defined and how operators to use operators. The abstract syntax is based on the EBNF concrete syntax defined in Listing 23.

Notice we do not explicitly represent the enumerations, in order to present a cleaner diagram. The enumeration “UnaryOperator”, type of the EAttribute labeled “operator”, owned by the EClass “UnaryOperation”, has the literals set on “unOp”. Analogously, “BinaryOperator” corresponds to “binOp”, “CompareOperator” to “compareOp”, “Quantifier” to “quant” and “Multiplicity” to “mult”.

Listing 23. Alloy's expression definition in EBNF.

```

expr ::= const | qualName | @name | this |
       unOp expr | expr binOp expr | expr arrowOp expr |
       expr [ expr,* ] |

```

```

expr [! | not] compareOp expr |
expr (=> | implies) expr else expr |
let letDecl, + blockOrBar |
quant decl, + blockOrBar |
{ decl, + blockOrBar } |
( expr ) | block

const ::= [-] number | none | univ | iden

unOp ::= ! | not | no | mult | set | # | ~ | * | ^
binOp ::= || | or | && | and | <=> | iff | => | implies | & | + | - | ++ | <: | :> | .

arrowOp ::= [mult | set] -> [mult | set]

compareOp ::= in | = | < | > | =< | >= |

letDecl ::= name = expr

block ::= { expr* }

blockOrBar ::= block | bar expr

bar ::= |

quant ::= all | no | sum | mult

mult ::= lone | some | one

qualName ::= [this/] (name /)* name

```

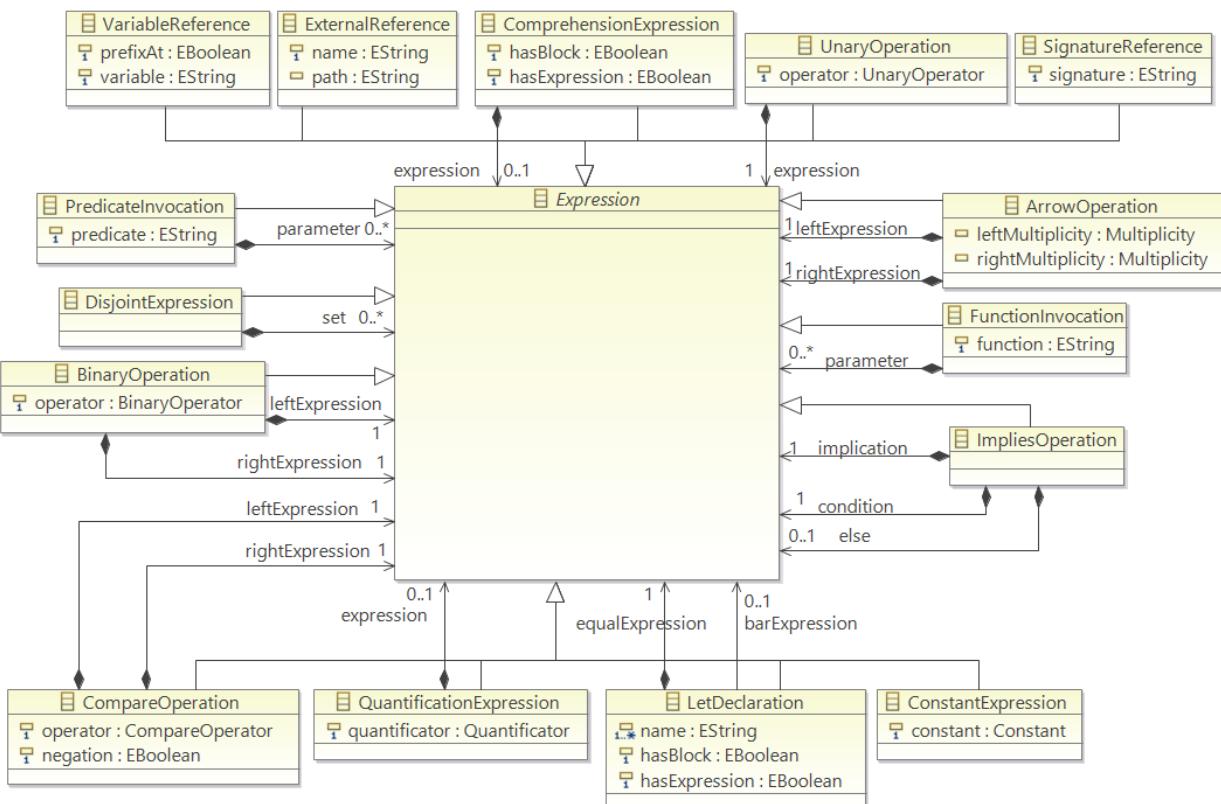


Figure 61. Alloy meta-model fragment: expressions.

8.4.2 User Interface

The transformation is available as an OLED component. Figure 62 show the user interface when calling the transformation function. We highlight the most noteworthy implementation features in red: the shortcuts for model and diagram simulation, the element selection tree, the simulation parameters and the transformation of OCL constraints.

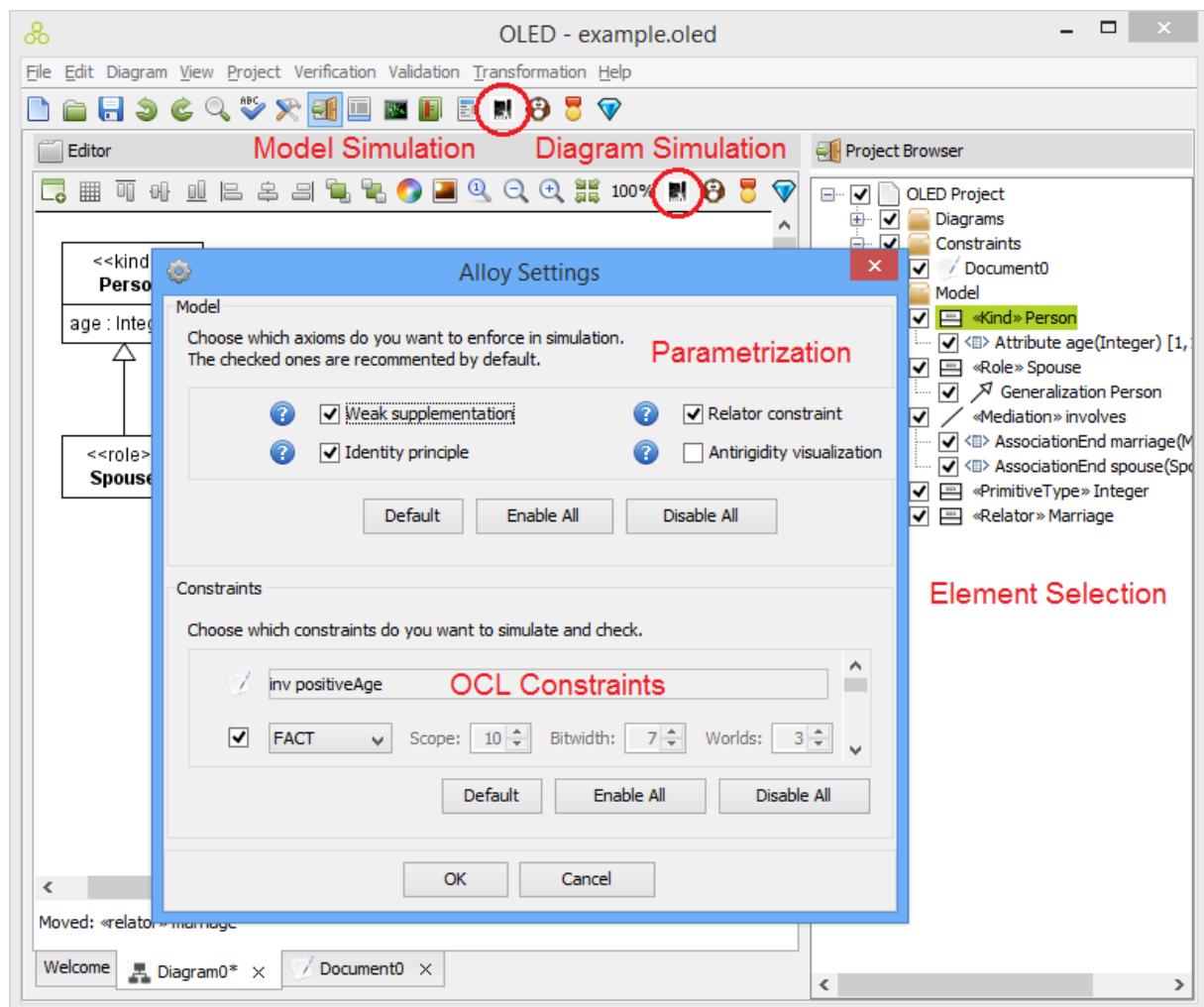


Figure 62. Simulation component within OLED.

In order to provide flexibility when simulating models, we implement two complementary commands: a model simulation, which only transforms to Alloy the checked elements in the element selection tree; and a diagram simulation, which only takes to Alloy the elements represented in the diagram.

Aiming to maintain as much integrity as possible when simulating partial models, both commands call an algorithm to auto-select mandatory dependencies automatically. For instance, if one selects an association, the algorithm selects all association ends, as well as their respective types. Furthermore, if one selects a generalization, the algorithm checks both child and parent classes. Moreover, if one selects attributes to simulate, the algorithm will select both the owner class and the attribute's type. This algorithm, however, does not guarantee (nor it intends to) that the resulting selection will be syntactically valid.

As we discussed in Section 3.2.5, we redesigned the transformation so it is able to cope with partial models. To achieve that, we propose four parameters that instruct the transformation to generate (or not) particular code structures. The tool presents them as four axioms users can choose to enforce. By default, the identity principle (that states that every individual must have a defined identity), the weak supplementation (every whole must have at least two parts) and the relators rule (every relator must mediated at least two individuals) are set to true. Anti-rigidity, on the other hand, is set to false.

Whenever a user calls a transformation command, the tool checks whether or not is interesting to disable one of the three enforced axioms. If there is a class stereotyped as subkind, role or phase that does not inherit an identity principle from a substance sortal, or a category, roleMixin or Mixin, which does not have any sortal class as descendant, the tool will instruct users to disable the identity principle axiom, otherwise the simulation will only show empty extensions for the identity-less classes. In the same way, if there is a whole composed at most by one part or a relator mediating at most one individual, the tool suggest disabling the weak supplementation and the relator axioms.

Another interesting transformation component, but which is not part of this research, is the transformation of OCL constraints, which is compliant with our Alloy mapping. In the simulation dialog, the tool prompts the user with the specified constraints, which he can choose to include in the Alloy generation. For more details of the OCL transformation, see (GUERSON; ALMEIDA; GUIZZARDI, 2014).

Moreover, we implemented a feature that is not visible in the user interface: the way we handle naming in Alloy. OntoUML models impose no constraints regarding element's names. Two or more classes, associations, attributes or associations ends can have the same name. They can even have empty or null names, or even names composed by special characters, like “#” or “!”. Regardless, Alloy has much more restrictive naming directives. For instance, signatures must be uniquely named and start with letters or an underscore (_). The same thing goes for functions, predicates, facts and assertions. Furthermore, as any textual language, Alloy defines reserved keywords, such as “sig”, “some”, “univ” and so on.

To deal with the naming differences between OntoUML and Alloy, we implement an alias-based work around. Each named element in the OntoUML model receives a unique alias that is unique and valid in Alloy. The alias generator performs the following actions: assign numbers to elements with repeated names, remove special characters (e.g. empty space, punctuation), include an underscore in reserved words, and adds an underscore for names beginning with numbers.

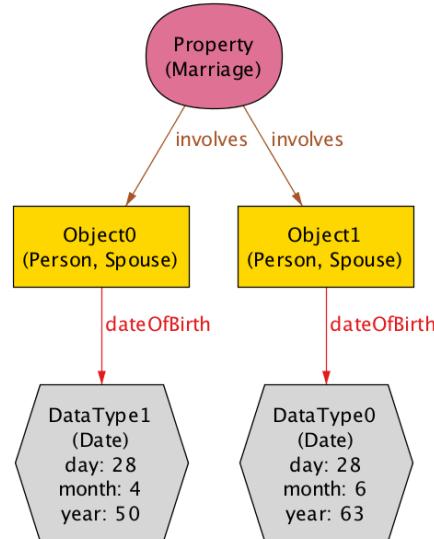


Figure 63. Custom theme: yellow boxes for objects, red ellipses for properties and grey hexagons for datatypes.

Lastly, we configure a custom theme for the instances generated by the Alloy Analyzer. We assigned a shape and a color for each ultimate meta-type: we represent objects (individual that instantiate kinds, quantities, collectives, subkinds, roles, phases, categories, roleMixins or mixins) with yellow boxes; properties (individuals that instantiate relators, modes or qualities) are represented as red ellipses; and lastly,

datatypes and primitive types are represented as light grey hexagons. Figure 63 illustrates an application of the theme.

8.5 THE ANTI-PATTERN COMPONENT

Once again, we reinstate that the ultimate goal in this research is to provide accessible alternatives for modelers to validate their ontologies without any additional training in special methods, tools or techniques. With that in mind, we adopted a strategy for managing anti-patterns that consists of three basic steps: automatic detection, guided analysis and automatic refactoring.

In order to relieve modelers from learning all anti-pattern structures and manually inspecting occurrences in their models, we implemented a component on OLED that does that automatically. Users can request an anti-pattern inspection on a particular diagram or on an arbitrary selection of elements, in the same way they did for the simulation. Moreover, as shown in Figure 64, the might instruct the tool to inspect the model selection for only a subset of the defined anti-patterns.

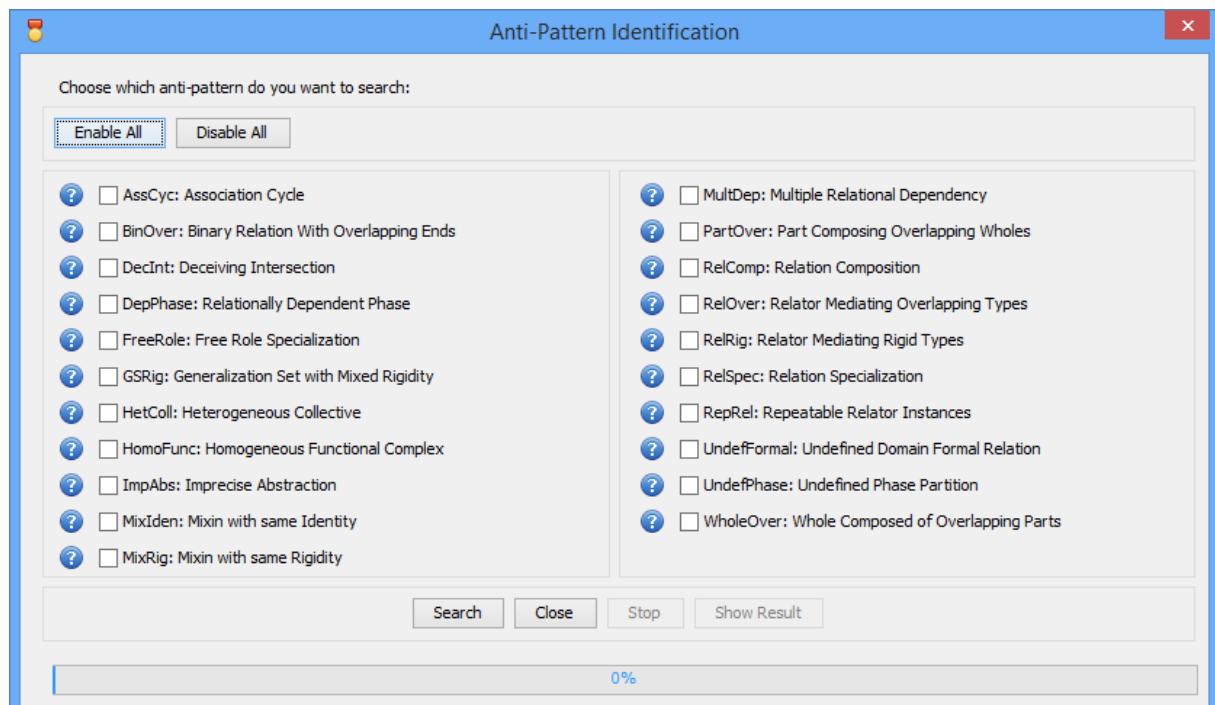


Figure 64. Anti-pattern identification dialog on OLED.

In the sequence, an additional dialog lists all occurrences identified for the selected anti-patterns, as depicted in Figure 65. We list the results using three columns:

- “Name”, which corresponds to a short description of the most relevant elements that characterize the anti-pattern;
- “Type”, that provides the acronym for the anti-pattern type; and
- “Status”, a binary property that can be set as “Opened”, if the respective occurrence still has not been analyzed, and as “Fixed”, if the occurrence is successfully analyzed.

The “Find” text field on the upper left corner allows users to search for occurrences, using as a reference the “Name” column. This is particularly useful when users perform detection on larger models, which results in a substantial number of occurrences. At this point, users can perform two actions for an anti-pattern occurrence: they can choose to analyze or ignore the occurrence.

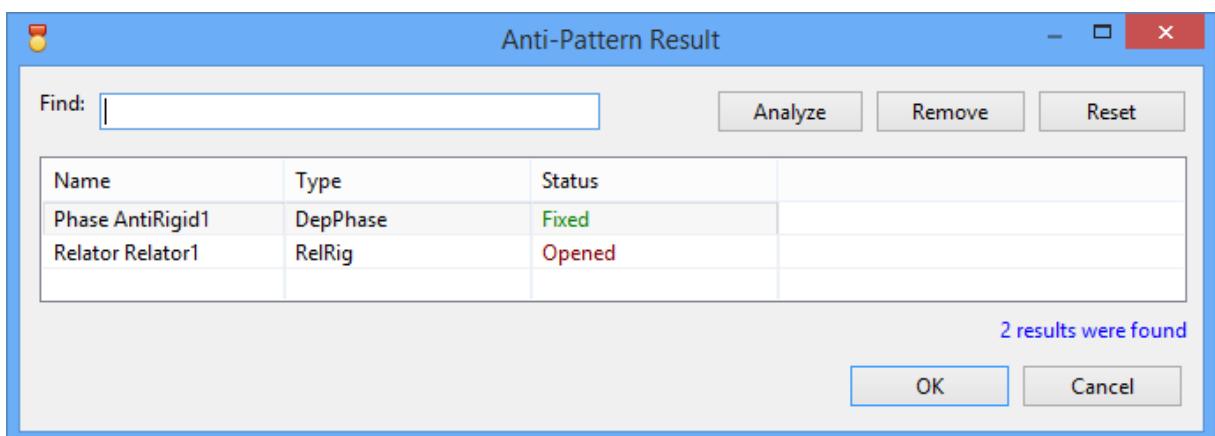


Figure 65. Anti-pattern result dialog in OLED.

The “Analyze” button gives rise to the second step of our strategy: the guided analysis. As we previously discussed, anti-patterns, in the sense that we use them, do not necessarily imply in domain misrepresentations. Furthermore, our anti-patterns are not constructions that should be discouraged, as seen in the context of software development (KOENIG, 1995). So, in order to decide whether a particular occurrence implies in undesired consequences or not, a modeler must reason about it. To support this process, we implemented a wizard for each anti-pattern, which details the elements that participate in the anti-pattern occurrence, provides theoretical notions

when necessary, and makes a series of questions, which lead to the appropriate solutions.

Every wizard starts with a page like the one in Figure 66. The dialog's title identifies the anti-pattern type. In this figure, an occurrence of the Relator Mediating Rigid Types (RelRig) occurrence is under analysis. It also provides the description of the elements that participate in the anti-pattern. In the example, a relator entitled "Marriage", connected to a rigid type, labeled "Marriage Certificate". Furthermore, it provides a general description of the anti-pattern type, in the example: *This anti-pattern occurs when a «relator» is connected through a «mediation» association to at least one rigid object type, stereotyped as «kind», «quantity», «collective», «subkind» or «category».*

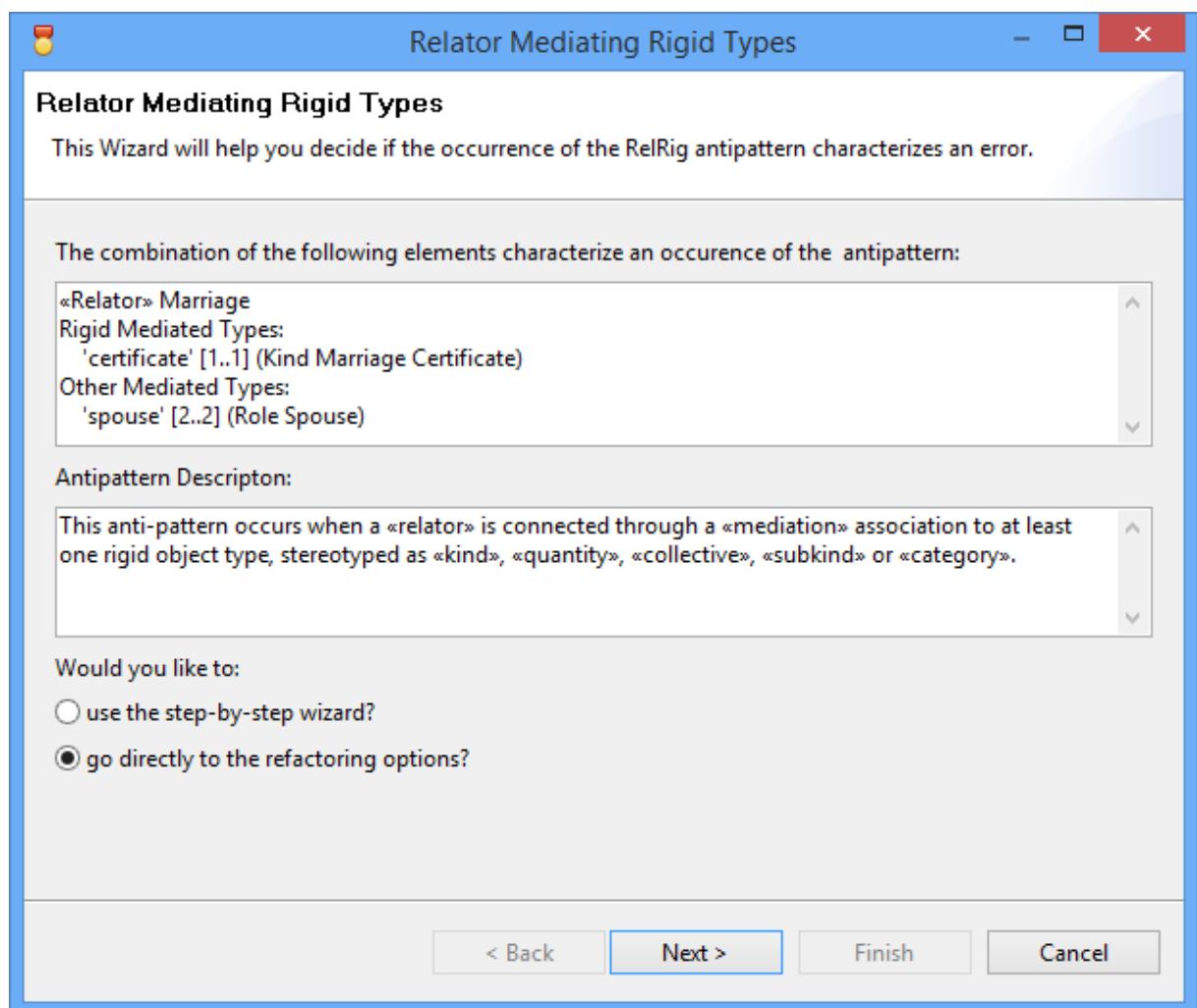


Figure 66. RelRig's initial anti-pattern wizard page.

The initial page of the wizards always asks the users if they want to go through a systematic analysis or choose directly a refactoring option. This question is included to

make the tool attractive for both novice and expert users. A user that is not familiar with the anti-pattern's structure and/or the possible consequences will likely go through the wizard, whilst another that already used the tool might already know how to fix his/her model and is more concerned with efficiency.

A novice user that opted to go through the RelRig's step-by-step wizard would reach the page depicted in Figure 67. In this page, the tool asks the user the first question of the wizard, which aims to verify if the rigid type connected to the relator, is indeed rigid, as the stereotype's choice suggests. In this case, by answering "Yes", the user already identifies a modeling error and the wizard already proposes a refactoring plan. If the answer is "No", the analysis continues with other questions.

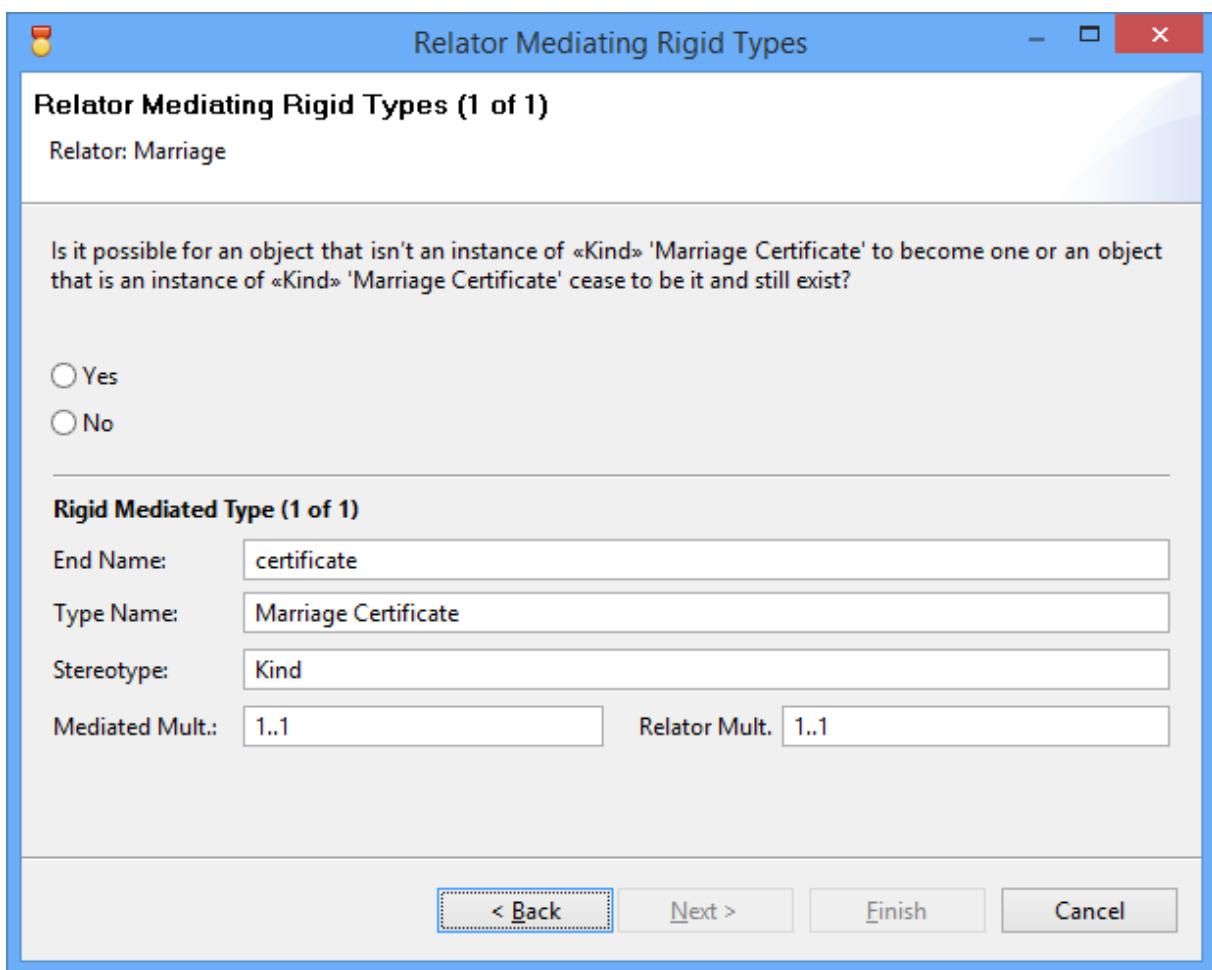


Figure 67. RelRig's wizard page example

An anti-pattern analysis always ends with one of the following conclusions: the occurrence characterizes a modeling problem fixable by a pre-defined refactoring plan or the identified structure is correct, and the occurrence is a false alarm. The tool

presents the analysis results on the finishing page, as depicted in Figure 68. In this stage, the last step of our anti-pattern strategy comes into play: the automatic refactoring. In the figure, the tool is informing the user that the provided answers led to conclusion that a refactoring is required, identified in the central box. In the figure, the tool is informing the user that the stereotype of the class labeled as “Marriage Certificate”, will change from Kind to Role. At this point, the user is still able to go back and revisit her answer, in case the proposed action does not satisfy her.

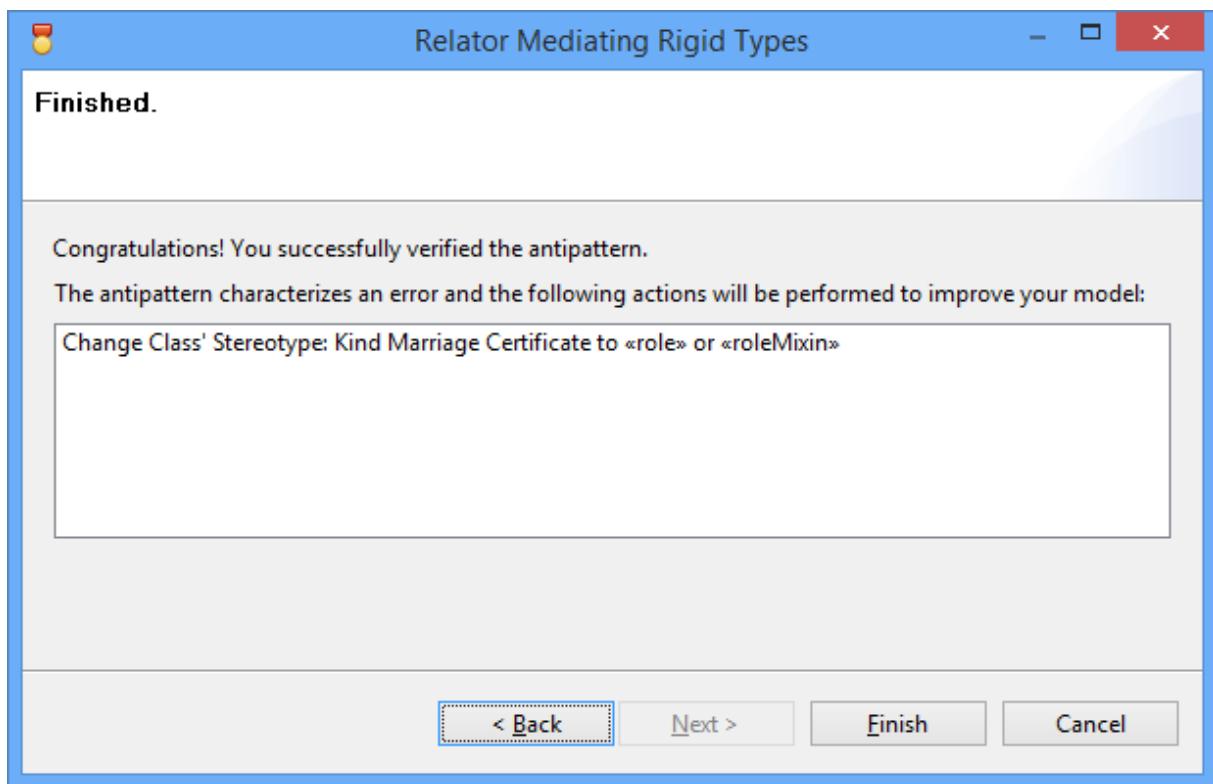


Figure 68. RelRig's finishing page.

9 CONCLUSIONS

9.1 CONTRIBUTIONS

This thesis contributes to the theory and practice of ontology-driven conceptual modeling.

The main theoretical contributions are:

- We offer an improved *transformation from OntoUML to Alloy* (discussed in Chapter 3), redesigned after a careful analysis of the consequences of the mappings defined by the previous approaches. Furthermore, we assigned parameters for the transformation that enabled it to cope with partial models, a demand identified in practice. The overall result is a much more flexible and useful validation tool than the previous initiative.
- A set of 20 *simulation scenarios* that make the simulation-based approach available accessible even to an untrained audience, detailed in Chapter 3. These scenarios are mainly of two types: branch structures, which allow the definition of the generated world structure; and the content scenarios, which allow defining characteristics regarding the individuals that exist in a world or amongst worlds. The content scenarios are particularly useful for checking conceptual model properties, like strong and weak finite satisfiability, class liveness and minimum and maximum association multiplicities.
- A *semantic anti-pattern catalogue* composed of 21 semantic anti-patterns, whose use in model validation improves the ontology quality, particularly regarding the precision, accuracy, scope and ontological classification measures. We detail this catalogue in Chapter 5 and validate it in Chapter 7, through an empirical experiment conducted in the context of an industrial project in partnership with the Brazilian government. In addition to improving model quality, anti-patterns showed the benefit of improving validation efficiency, since they were able to predict recurrently the right solutions for anti-pattern occurrences.

The activities performed in this research also generated some indirect contributions to the theory of ontology-driven conceptual modeling in general, and for OntoUML model development, particularly:

- We empirically defined an *approach to identify recurrent problematic modeling decisions* in ontology-driven conceptual models, mainly based on visual model simulation (see Section 6.2).
- We provide strong evidence that *model simulation is useful for model validation*. This contribution is a side effect of using the simulation we redesigned in the beginning of this thesis to uncover anti-patterns. Since logical errors were systematically identified, especially regarding model under-constraining, we corroborate the approach usability.
- By reasoning about the problems that anti-pattern occurrences might give rise to and conceiving pre-defined refactoring plans, we *identified and formalized a number of domain-independent modeling patterns*, like the role specialization patterns, e.g. sub-role, role of role and derived-role (see Section 5.5) and the phase partition patterns, particularly the Intentional Partition (see Section 5.20). Furthermore, we also identified Rule-patterns for specific modeling structures, like exclusive roles of a relator, or exclusive parts of a whole.
- The empirical study we performed for validating the anti-pattern catalogue brought an unexpected result: *the identification of language expressivity demands* (see the discussion in Section 7.2). For instance, the analysis of the UndefFormal occurrences evidenced that the users require further classifications for formal relations. Furthermore, other anti-patterns identified the need to explicitly represent qua-individuals, events and higher-order universals, as well as including in the language a constitution relation, which holds between qualities and objects.

The technological contributions are two-fold:

- A fully functional *implementation of the redesigned transformation from OntoUML to Alloy*, distributed as an OLED component (see Section 8.4). The tool features many useful functionalities, like automatic naming treatment, manual parameterization and the selection of model elements to simulate.

- A fully functional *computational environment that automates anti-pattern management*, by automatically detecting occurrences, guiding users through their analysis in order to select the appropriate refactoring solution (if necessary), and automatically executing the selected solution (see Section 8.5).

Lastly, we contribute to other fellow researchers by assembling an OntoUML model repository (we describe it in Section 6.1). There is no other available and it can be very useful for in future researches, like further understanding how the language is used, identifying additional modeling patterns, “elephant paths” in the language, and even additional recurrent modeling problems.

9.2 DISCUSSION

In Chapter 1, we stated that our goal is to develop an ontology validation framework that aid modelers in systematically producing higher quality ontologies, without requiring costly training in particular techniques. Now, we discuss to what extent our proposals meet the requirements specified by the goal.

Regarding the capability of systematically improving model quality, we argue that all proposals manage to do that. First, the simple fact that we were able to uncover an anti-pattern catalogue using model simulation is a strong corroboration that the simulation successfully contributes for identifying modeling errors and thus improving model quality. In addition, the properties related to the validation scenarios show that the modelers are able to verify useful model properties, like strong and weak satisfiability, class liveness and minimum cardinalities. By identifying satisfiability issues, one certainly improves model quality, whilst “dead” class identification highlights over-constraining problems. Lastly, we find strong evidence that the anti-patterns contribute to model validation in the empirical study – from the 1475 identified occurrences, modelers considered 794 as errors, what corresponds to 53.8% of the cases.

We also stated that our validation framework should be accessible to users without costly learning efforts. Although this is a subjective feature, we argue that it is safe to say that our proposals relieve users from heavy training in particular techniques. First,

even though knowing Alloy’s logic and syntax allows more flexibility on validation, we relieved users from this “obligation” with the simulation scenarios. With them, users are able to simulate models without even knowing what Alloy is. With further tool support, it will be even possible to simulate models without even seeing the generate Alloy code. In the previous approaches, in order to use any feature of the simulation, one had to learn two things: the Alloy language and the transformation’s mappings.

Furthermore, the tool support we developed for anti-patterns also requires little previous knowledge from users. First, by automatically detecting occurrences, it relieves users to learn the anti-pattern structure definition. Not only that but users are taught the structure “on the fly”, through the tool explanation of possible problem’s source. In the sequence, the wizard implementation mitigates relieves the problem of reasoning about the possibilities implied by a given structure. Finally, by assigning a refactoring plan to a set of answers and running it automatically, users do not need to think about the solution and neither run it themselves.

9.3 SHORTCOMINGS

In this section, we elaborate on shortcomings and limitations of the techniques and tools proposed in this thesis.

Regarding model validation through simulation, as proposed in Chapter 3, we identify two limitations inherent from Alloy: the limitation of the size of the model one can simulate and the respective number of instances the analyzer is capable to deal with. Regardless of performance adjustments, the simulation will always suffer from these problems because they come from the way the analyzer generates instances: solving satisfiability problems, a problem well known as NP-Complete. By not being able to simulate the model as a whole, it requires users to fragment their models. Validating sub-models is not a guarantee that the model as a whole will not have problems.

The simulation scenarios pose a trade-off between usability and expressivity. On one hand, our solution makes Alloy accessible for untrained users, while on the other, it limits the number of expressions modelers can form and, thus, the properties they can checked.

Moreover, we recognize that, for each anti-pattern, we did not cover every single possible refactoring alternative. Instead, we focused on recurrent ones. By doing that, we are able to balance the time required to analyze an anti-pattern occurrence and the anti-pattern solution predictability rate.

On the technological level, we identify two limitations regarding anti-pattern tooling. First, some anti-pattern occurrences, even though are properly analyzed and fixed, keep showing up in the tool as possible mistakes. The reason is that our identification algorithms only inspect the model structure, leaving the OCL rules aside. Thus, the tool “thinks” that nothing has changed when rules are created.

The second anti-pattern tool limitation regards the efficiency of some algorithms designed to for anti-pattern detection. Most of the time we were able to develop very fast algorithms, that even in ontologies like the MGIC one, with 3800 classes and 1800 associations, all occurrences were identified under 5 seconds. Nonetheless, for a few anti-patterns, like AssCyc, RelComp and WholeOver, the identification did not perform as well in large models. Their performance on small and medium models (with at most 100 classes) however, is still acceptable.

9.4 RELATED WORKS: ANTI-PATTERNS

Since the Koenig’s original proposal (KOENIG, 1995), the concept of anti-pattern has been applied in a variety of fields other than software design. To the extent of our knowledge, however, there is no other application of anti-patterns in ontology-driven conceptual modeling in general, and particularly, to OntoUML. For that reason, we now compare our anti-pattern proposal and catalogue proposal to other works that also apply anti-patterns (or similar ideas) in the context of structural conceptual modeling and semantic web ontology validation.

For each related work, we discuss their definition of anti-pattern (or similar concept), the language for which its catalogue was defined for, the proposed anti-pattern classifications, whether or not refactoring alternatives are provided, the formalism use to define them, the intended applications, the elicitation methods and the tool support available.

9.4.1 OWL anti-patterns

The research reported in (CORCHO; ROUSSEY; BLAZQUEZ, 2009; ROUSSEY; ZAMAZAL, 2013; TAHWIL, 2010) define anti-pattern as patterns that are commonly used by domain experts in their OWL implementations and that normally result in inconsistencies or modeling errors. Furthermore, the authors state that anti-patterns come from a misuse and misunderstanding of description logics expressions by ontology developers. The research generated an anti-pattern catalogue for OWL ontologies, formalized in description logics, and that classifies anti-patterns in three exclusive categories:

- *logical anti-patterns*, that represent errors that reasoners detect i.e., consistency problems;
- *cognitive anti-patterns*, that represent possible modelling errors that are not detected by reasoners; and
- *guidelines*, that stand for complex expressions used in an ontology component definition that are correct from a logical point of view, but in which the ontology developer could have used other simpler alternatives for encoding the same knowledge.

The reported elicitation method (CORCHO; ROUSSEY; BLAZQUEZ, 2009) is a case study conducted as an interactive collaboration with a domain expert in the development of an ontology for a Spanish governmental institution, named HydrOntology. The authors propose anti-patterns as means to improve efficiency on ontology debugging, a sort of validation activity. The tool that supports the approach is named Apero (AntiPatternExtRactiOn) and is developed as a Protégé plugin (TAHWIL, 2010), a very popular tool for building OWL ontologies.

Our view on anti-patterns is quite similar to theirs: modeling structures that might indicate problems. However, we are concerned with different types of problems. We are not concerned with inconsistency issues because OntoUML's design prevents them. Moreover, we are also not concerned with guidelines because we want to improve ontology precision and accuracy, not readability or maintainability. Therefore, the cognitive anti-patterns are the closest to our proposal. The difference is that we take

into consideration all the ontological distinctions contained in OntoUML's metamodel, whilst they are limited to description logics operators.

9.4.2 OOPS!

The research reported in (POVEDA; SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2010a, 2010b; POVEDA-VILLALÓN; SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2012), uses two concepts: anti-pattern and common pitfall. Anti-patterns are designs that matches an Ontology Design Pattern (ODP) (GANGEMI, 2005) but this design is not a suitable solution to the modelling problem and also designs that, although do not match an ODP, could be solved by an existing design pattern. Common pitfalls, on the other hand, are an unsuitable solution to the modelling problems that unsolvable with ODPs. As the OWL Anti-patterns, they also see these recurrent problems as misunderstanding and misusing the description logics constructs.

The authors classify the common pitfalls, identified in RDF and OWL ontologies, from two different perspectives. First, w.r.t the type of problems they can cause, which leads to the classifications: structural, functional and usability; second, regarding the quality criteria they affect: consistency, completeness and conciseness.

The pitfall catalogue was identified by manually inspecting models that were developed by master students (POVEDA; SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2010b) in the context of a masters ontology engineering course. Their study analyzed 26 models.

A web-based tool named “OOPS! An Ontology Pitfall Scanner” supports their approach (POVEDA-VILLALÓN; SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2012). The authors also distribute the tool as a web service, pluggable to Protégé²⁰.

Our approach requires anti-patterns to have a finite and identifiable structure. In contrast, many pitfalls proposed in OOPS do not share this property. Some depend on class names, like merging to concepts in one, which analyzes if there is an “and” or an “or” on the class name. Moreover, some pitfalls are exclusively dependent on domain knowledge, like the improper creation of equivalent classes for synonyms. A priori,

²⁰ <http://protege.stanford.edu/>

there is no indication that two equivalent classes are synonyms. We removed from our catalogue the relator dilemma (Section 5.23) because there is no possible structural identification algorithm.

9.4.3 Meta-modeling anti-patterns

The series of publications (ELAASAR; BRIAND; LABICHE, 2010, 2011, 2013) propose not only a language for specifying patterns in MOF-based languages, named Visual Pattern Modeling Language (VPML) , but also a library of meta-modeling anti-patterns. We compare our proposal to this work because meta-modeling is still modeling.

The authors classify the anti-patterns as:

- *well-formedness*, that refer to integrity constraints which may or may not be included in the language as syntactical constraints, but that are always errors;
- *semantic*, a category of anti-patterns that defines UML designs that are well-formed syntactically but that could be problematic semantically when used for meta-modeling; and
- *conventional*, a category that captures common violations in particular naming and documentation conventions.

As in the other proposals, the authors include as anti-patterns decisions that are always wrong (well-formedness) and modeling guidelines (conventional). These two categories are out of the scope of our work, as previously discussed. The semantic ones resemble our proposal, but they focus much more on syntactical features, whilst we are concerned with the conceptualization one is trying to formalize. Furthermore, the authors do not propose any refactoring plan or analysis instruction, only the identification of the modeling structure.

9.4.4 UML anti-patterns

The research reported in (BALABAN; MARAEE; STURM, 2010; BALABAN et al., 2014) discusses anti-patterns in the context of UML class diagrams. The research

focus is on designing a model correctly, and not designing the correct model. In other words, they are concerned with making a model work. The model aspects analyzed by the proposed anti-pattern catalogue are concerned with correctness issues, namely consistency and finite satisfiability, and particular types of quality criteria, completeness and conciseness.

The authors specify their anti-pattern catalogue using their own language, named Pattern Diagram Class (PDC) (BALABAN et al., 2014), but they do not provide any tool support for the identification of the proposed anti-patterns in UML models. Differently from all other approaches, they envision anti-patterns as means for teaching modelers and improving their modeling skills. Moreover, the authors do not discuss how they came up with their anti-pattern catalogue.

Our work differs from theirs regarding the type of feature we are interested. One can even say that they are more concerned with verification, whilst we focus on validation.

9.4.5 General Remarks

Notice that, in all approaches, some anti-patterns always imply in modeling mistakes. The UML consistency anti-patterns, the OWL logical ones and the meta-modeling well-formedness issues all point out to definitely incorrect models. As our anti-pattern definition states, that sort of problem is not within the scope of this work. Moreover, it is not because of two reasons: firstly, because there are a lot less inconsistency possibilities in OntoUML. As discussed in Chapter 2, the foundational ontology from which OntoUML derives forbids many of such errors. Secondly, because we advocate that OntoUML should forbid, on a syntactical level, syntactical combinations that are always wrong.

Overall, one of the improvements we make on these proposals, considering the general application of anti-patterns in domain modeling, is the development of a wizard to guide users on the analysis of an anti-pattern occurrence. Because of that, we do not require users to study the anti-pattern catalogue before using the tool. The wizard is especially useful for those situations that are not always errors. In fact, only our proposal systematically provides multiple solutions for the same issue.

A recurrent difference from our work to all aforementioned proposals is that we are concerned with a special type of model quality: domain appropriateness, i.e., adequately capturing the conceptualization of a domain. Our goal is not to help modelers build the models correctly, but build the correct model for the domain at focus.

9.5 OPEN QUESTIONS

Quantity and quality related anti-patterns. Even though we proposed 21 semantic anti-patterns in this thesis, none of them validates the usage of quantities and qualities (both datatypes and proper qualities) in ontology-driven conceptual models. The little usage of classes stereotyped as Quantity in the models we were able to gather diminished the possibility of identifying anti-patterns about them. Because of that, we envision the execution of more directed empirical experiments, assigning domains that necessarily require the usage of quantities. The same is valid for quality usage.

Pro-active use of anti-patterns. During the development of the tools and the tests performed by associates, we detected that, after a while using the tool, modelers get so familiar with the anti-patterns that they already identify occurrences when modeling. Not only that but, at modeling time, they already know how to “fix” their decisions want to pro-actively enforce determined situations.

Instance-level modeling constructs. Instead of reacting to problems identified by the anti-patterns, we envision the possibility of preventing them. One way to do that is to include new modeling constructs in OntoUML, especially for the recurrent rule pattern identified by the anti-patterns, which identify instance-level constraints. For example, the RelOver anti-pattern generates rules that make the mediated types exclusive. Other possible construct would allow the specification of the number of times the same elements can be connected by different instances of the same relator type, as discussed in the RepRel anti-pattern. Overall, one must analyze the trade-off between language expressivity and complexity.

The relator dilemma. As we uncover the semantic anti-patterns, we identified a recurrent problem for which we could not assign a distinguished structure. We named

it the relator dilemma. Simply put, this error regards the representation of a concept as a relator when it is in fact an event, a normative description or even a material relation. We did not investigate this problem further, but understanding it better should facilitate envisioning ways for preventing it to happen.

Tool support for simulation scenarios. During this research, we were not able to develop tool support for the simulation scenarios. In order for a wider adoption of the simulation, a proper tool support for the scenarios is in order. With that, it is even possible to evaluate their usability through empirical tests and interaction with novice users.

Alternative validation proposals. In this thesis, we presented three complementary approaches for the validation of OntoUML models. Nonetheless, throughout this research we studied other approaches that would complement the existing ones. For instance, the model testing approach for conceptual schemas (TORT; OLIVÉ; SANCHO, 2011) can surely be adapted for OntoUML and be a very useful alternative to provide a validation mechanism for the ontology as a whole. Furthermore, the proposal for validation of UML models based on OCL constraints (QUERALT; TENIENTE, 2012) can also be an alternative to provide holistic validation for OntoUML models.

REFERENCES

- ALBUQUERQUE, A. **Desenvolvimento de uma Ontologia de Domínio para Modelagem de Biodiversidade.** Manaus: Universidade Federal do Amazonas, 2011.
- ALBUQUERQUE, A. F. DE; GUIZZARDI, G. **An Ontological Foundation for Conceptual Modeling Datatypes based on Semantic Reference Spaces** (R. Wieringa, S. Nurcan, C. Rolland, J.-L. Cavarero, Eds.). In: International Conference on Research Challenges in Information Science (RCIS). **In Proceedings...** Paris, France: IEEE Press, 2013. P. 1–12.
- ANDONI, A.; DANILIUC, D.; KHURSHID, S.; MARINOV, D. **Evaluating the Small Scope Hypothesis.** Cambridge, USA: MIT Laboratory for Computer Science, 2002. 12 p.
- AZEVEDO, C. L. B.; ALMEIDA, J. P. A.; VAN SINDEREN, M.; QUARTEL, D.; GUIZZARDI, G. **An Ontology-Based Semantics for the Motivation Extension to ArchiMate.** In: International Enterprise Distributed Object Computing Conference (EDOC). **In Proceedings...** Helsinki, Finland: IEEE Computer Society, ago. 2011. P. 25–34.
- BALABAN, M.; MARAEE, A.; STURM, A. Management of Correctness Problems in UML Class Diagrams Towards a Pattern-Based Approach. **International Journal of Information System Modeling and Design**, v. 1, n. 4, p. 24–47, 2010.
- BALABAN, M.; MARAEE, A.; STURM, A.; JELNOV, P. A pattern-based approach for improving model quality. **Software & Systems Modeling**, 2014.
- BARCELOS, P. P. F.; GUIZZARDI, G.; GARCIA, A. S.; MONTEIRO, M. **Ontological Evaluation of the ITU-T Recommendation G.805.** In: International Conference on Telecommunications (ICT). **In Proceedings...** Ayia Napa, Cyprus: IEEE Press, 2011. P. 266–271.
- BARCELOS, P. P. F.; GUIZZARDI, R. S. S.; GARCIA, A. S. **An Ontology Reference Model for Normative Acts.** In: Seminário de Pesquisa em Ontologias no Brasil (ONTOBRAS). **In Proceedings...** Belo Horizonte, Brazil: CEUR-WS.org, 2013. P. 35–46.
- BARCELOS, P. P. F.; SANTOS, V. A. DOS; SILVA, F. B.; MONTEIRO, M.; GARCIA, A. S. **An Automated Transformation from OntoUML to OWL and SWRL** (M. P. Bax, M. B. Almeida, R. Wassermann, Eds.). In: Seminário de Pesquisa em Ontologias no Brasil (ONTOBRAS). **In Proceedings...** Belo Horizonte, Brazil: CEUR-WS.org, 2013. P. 130–141.
- BASTOS, C. A. M.; REZENDE, L.; CALDAS, M.; GARCIA, A. S.; MECENA FILHO, S.; SANCHEZ, M. L.; CASTRO JUNIOR, J. DE L. P. **Building up a Model for Management Information and Knowledge : The Case-Study for a Brazilian**

Regulatory Agency. In: International Workshop on Software Knowledge (SKY). **In Proceedings...** Paris, France: 2011.

BENEVIDES, A. B. **A Model-based Graphical Editor for Supporting the Creation, Verification and Validation of OntoUML Conceptual Models.** Vitória, Brazil: [s.n.]. p. 259

BENEVIDES, A. B.; GUZZARDI, G.; BRAGA, B. F. B.; ALMEIDA, J. P. A. Validating Modal Aspects of OntoUML Conceptual Models Using Automatically Generated Visual World Structures. **Journal of Universal Computer Science**, v. 16, n. 20, p. 2904–2933, 2010a.

BENEVIDES, A. B.; GUZZARDI, G.; BRAGA, B. F. B.; ALMEIDA, P. A. Assessing Modal Aspects of OntoUML Conceptual Models in Alloy. **Journal of Universal Computer Science**, v. 16, n. 20, p. 2904–2933, 2010b.

BORDBAR, B.; ANASTASAKIS, K. **UML2Alloy: A Tool For Lightweight Modelling of Discrete Event Systems.** In: IADIS International Conference on Applied Computing. **In Proceedings...** Algarve, Portugal: IADIS, 2005. P. 209–216.

BRAGA, B.; ALMEIDA, J.; GUZZARDI, G.; BENEVIDES, A. Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. **Innovations in Systems and Software Engineering**, v. 6, n. 1, p. 55–63, 2010.

BRASIL. Lei de Acesso à Informação. Law nº 12.527, November 18th, 2011.

BROWN, W. J.; MALVEAU, R. C.; MCCORMICK III, H. W.; MOWBRAY, T. J. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.** New York, USA: John Wiley & Sons, 1998. p. 336

CALHAU, R. F.; FALBO, R. DE A. **An Ontology-Based Approach for Semantic Integration.** In: International Enterprise Distributed Object Computing Conference (EDOC). **In Proceedings...** IEEE, out. 2010. P. 111–120.

CALHAU, R. F.; FALBO, R. DE A. **A Configuration Management Task Ontology for Semantic Integration.** In: ACM Symposium on Applied Computing (SAC). **In Proceedings...** SAC '12. New York, USA: ACM, 2012. P. 348–353.

CAROLO, F.; BURLAMAQUI, L. **Improving Web Content Management with Semantic Technologies.** In: SemTech. **In Proceedings...** San Francisco: 2011.

CARRARETTO, R. **A Modeling Infrastructure for OntoUML.** Vitória, Brazil: Universidade Federal do Espírito Santo, 2010.

CARRARETTO, R.; ALMEIDA, J. P. A. **Separating Ontological and Information Modeling Concerns Towards a Two-Level Model-Driven Approach.** In: International Workshop on Models and Model-Driven Methods for Service Engineering (3M4SE). **In Proceedings...** Beijing, China: IEEE Computer Society, 2012. P. 29–37.

CHEN, P. P. The Entity-Relationship Model: Toward a Unified View of Data. **ACM Transactions on Database Systems**, v. 1, p. 9–36, 1976.

CORCHO, O.; ROUSSEY, C.; BLAZQUEZ, L. M. V. **Catalogue of Anti-Patterns for formal Ontology debugging**. In: AFIA 2009: Atelier Construction d'ontologies : vers un guide des bonnes pratiques. **In Proceedings...** Hammamet, Tunisie: 2009.

COSTAL, D.; GÓMEZ, C.; GUIZZARDI, G. **Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML**. In: International Conference on Conceptual Modeling (ER). **In Proceedings...** Brussels, Belgium: 31 out. 2011. P. 189–203.

CRUZ, S. M. S. DA; CAMPOS, M. L. M.; MATTOSO, M. **A Foundational Ontology to Support Scientific Experiments** (A. Malucelli, M. Peixoto, Eds.). In: ONTOBRAS/MOST. **In Proceedings...** Recife, Brazil: CEUR-WS.org, 2012. P. 144–155.

DIJKSTRA, E. W. The Humble Programmer. **Communications of the ACM**, v. 15, n. 10, p. 859–866, 1972.

ELAASAR, M.; BRIAND, L. C.; LABICHE, Y. **Metamodeling Anti-Patterns**. Available at: <https://sites.google.com/site/metamodelingantipatterns>.

ELAASAR, M.; BRIAND, L. C.; LABICHE, Y. **Domain-Specific Model Verification with QVT**. In: European Conference on Modelling Foundations and Applications (ECMFA). **In Proceedings...** Birmingham, UK: Springer Berlin Heidelberg, 2011. P. 282–298.

ELAASAR, M.; BRIAND, L. C.; LABICHE, Y. VPML: an approach to detect design patterns of MOF-based modeling languages. **Software & Systems Modeling**, p. 1–30, 2013.

EVERMANN, J.; WAND, Y. Ontology based object-oriented domain modelling: fundamental concepts. **Requirements Engineering**, v. 10, n. 2, p. 146–160, 13 jan. 2005.

FALBO, R. DE A.; MENEZES, C. S. DE; ROCHA, A. R. C. DA. **A Systematic Approach for Building Ontologies**. In: Ibero-American Conference on Artificial Intelligence (IBERAMIA). **In Proceedings...** Lisbon, Portugal: Springer Berlin Heidelberg, 1998. P. 349–260.

FERRANDIS, A. M. M.; LÓPEZ, O. P.; GUIZZARDI, G. **Applying the Principles of an Ontology-Based Approach to a Conceptual Schema of Human Genome**. In: International Conference on Conceptual Modeling (ER). **In Proceedings...** Hong Kong, China: Springer Berlin Heidelberg, 2013. P. 471–478.

FERREIRA, M. I. G. B. **Ontologia de Emergência no Apoio à Geração de Soluções de Variabilidade de Planos de Emergência (MSc. Thesis)**. Rio de Janeiro, Brazil: Univerisade Federal do Rio de Janeiro, 2013.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 1. ed. Reading, USA: Addison-Wesley, 1999. p. 464

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston, USA: Addison-Wesley, 1994. p. 416

GANGEMI, A. **Ontology Design Patterns for Semantic Web Content**. In: International Semantic Web Conference (ISWC). **In Proceedings...** Galway, Ireland: Springer Berlin Heidelberg, 2005. P. 262–276.

GANGEMI, A.; CATENACCI, C.; CIARAMITA, M.; LEHMANN, J. **Ontology evaluation and validation: An integrated formal model for the quality diagnostic task**. Rome, Italy: Laboratory of Applied Ontologies (CNR), 2005. 53 p. Available at: http://www.loa-cnr.it/Files/OntoEval4OntoDev_Final.pdf

GANGEMI, A.; CATENACCI, C.; CIARAMITA, M.; LEHMANN, J. **Modelling Ontology Evaluation and Validation** (Y. Sure, J. Domingue, Eds.). In: European Semantic Web Conference (ESWC). **In Proceedings...** Budva, Montenegro: Springer Berlin Heidelberg, 2006. P. 140–154.

GONÇALVES, B.; GUIZZARDI, G.; FILHO, J. G. P. Using an ECG reference ontology for semantic interoperability of ECG data. **Journal of Biomedical Informatics**, v. 44, n. 1, p. 126–136, 2011.

GONÇALVES, B.; GUIZZARDI, G.; GONÇALVES, J.; FILHO, P. **An Electrocardiogram (ECG) Domain Ontology**. In: Brazilian Workshop on Ontologies and Metamodels for Software and Data Engineering (WOMSDE). **In Proceedings...** João Pessoa, Brazil: 2007. P. 68–81.

GUARINO, N. **The Ontological Level** 1994. P. 443–456.

GUARINO, N. **Formal Ontology and Information Systems**. In: Formal Ontology and Information System (FOIS). **In Proceedings...** Amsterdam, The Netherlands: IOS Press, 1998. P. 3–15.

GUARINO, N.; WELTY, C. A. An Overview of OntoClean. In: STAAB, S.; STUDER, R. (Eds.). **Handbook on Ontologies**. Berlin: Springer Berlin Heidelberg, 2009. p. 201–220.

GUERSON, J. O.; ALMEIDA, J. P. A.; GUIZZARDI, G. **Support for Domain Constraints in the Validation of Ontologically Well-Founded Conceptual Models**. In: International Conference on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD). **In Proceedings...** Thessaloniki, Greece: Springer Verlag, 2014. P. 302–316.

GUIZZARDI, G. **Ontological Foundations for Structural Conceptual Models**. Enschede, The Netherlands: Centre for Telematics and Information Technology, University of Twente, 2005.

GUZZARDI, G. **The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited.** In: International Conference on Advanced Information Systems Engineering (CAiSE). **In Proceedings...** Amsterdam, The Netherlands: Springer Berlin Heidelberg, 2009. P. 94–109.

GUZZARDI, G. Theoretical foundations and engineering tools for building ontologies as reference conceptual models. **Semantic Web Journal**, v. 1, n. 1,2, p. 3–10, 2010.

GUZZARDI, G. **Ontological Foundations for Conceptual Part-Whole Relations: the Case of Collectives and Their Parts.** In: International Conference on Advanced Information Systems Engineering (CAiSE). **In Proceedings...** London, UK: Springer Berlin Heidelberg, 2011. P. 138–153.

GUZZARDI, G.; BAIÃO, F. A.; LOPES, M.; DE ALMEIDA FALBO, R. The Role of Foundational Ontologies for Domain Ontology Engineering: An Industrial Case Study in the Domain of Oil and Gas Exploration and Production. **International Journal of Information Systems Modeling and Design (IJISMD)**, v. 1, n. 2, p. 1–22, 2010.

GUZZARDI, G.; WAGNER, G. **Tutorial: Conceptual simulation modeling with Onto-UML.** In: Winter Simulation Conference (WSC). **In Proceedings...** Berlin, Germany: IEEE Computer Society, 2012. P. 1–15.

GUZZARDI, R. S. S.; FRANCH, X.; GUZZARDI, G.; WIERINGA, R. **Ontological Distinctions between Means-End and Contribution Links in the i* Framework** (W. Ng, V. C. Storey, J. Trujillo, Eds.). In: International Conference on Conceptual Modeling (ER). **In Proceedings...** Lecture Notes in Computer Science. Hong Kong, China: Springer Berlin Heidelberg, 2013. P. 463–470.

HALPIN, T.; MORGAN, T. **Information Modeling and Relational Databases.** 2. ed. Burlington, MA: Morgan Kaufmann, 2008. p. 942

HARTMANN, S. **Coping with Inconsistent Constraint Specifications.** In: International Conference on Conceptual Modeling (ER). **In Proceedings...** Yokohama, Japan: Springer Berlin Heidelberg, 2001. P. 241–255.

JACKSON, D. **Software Abstractions: Logic, Language, and Analysis.** Revised Ed ed. Cambridge, Massachusetts: The MIT Press, 2012. p. 354

JANSSEN, W.; MATEESCU, R.; MAUW, S.; FENNEMA, P.; STAPPEN, P. VAN DER. **Model Checking for Managers.** In: 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking. **In Proceedings...** London, UK: Springer-Verlag, 1999. P. 92–107.

KOENIG, A. Patterns and Antipatterns. **Journal of Object-Oriented Programming**, v. 8, n. 1, p. 46–48, 1995.

MARÍN, B.; GIACCHETTI, G.; PASTOR, O.; ABRAN, A. A Quality Model for Conceptual Models of MDD Environments. **Advances in Software Engineering**, v. 2010, p. 1–17, 2010.

MARTINS, A. F.; DE ALMEIDA FALBO, R. **Models for Representing Task Ontologies.** In: Workshop On Ontologies and Their Applications (WONTO). In Proceedings... Salvador, Brasil: CEUR-WS.org, 2008.

MONTEIRO, M.; GARCIA, A. S.; BARCELOS, P. P. F.; GUIZZARDI, G. Ontology Based Model for the ITU-T Recommendation G.805: Towards the Self-Management of Transport Networks. **International Journal of Computer Science and Information Technology**, v. 2, n. 2, p. 155–170, 2010.

MOODY, D. L.; SINDRE, G.; BRASETHVIK, T.; SØLVBÆRG, A. **Evaluating the Quality of Information Models: Empirical Testing of a Conceptual Model Quality Framework.** In: International Conference on Software Engineering (ICSE). In Proceedings... Portland, USA: IEEE Computer Society, 2003. P. 295–307.

MPOG. **Esboço de Modelagem Conceitual para Estruturas Organizacionais Governamentais Brasileiras e o SIORG.** 2011. 10 p.

MYLOPOULOS, J. Conceptual Modeling and Tellos. In: LOUCOPOULOS, P.; ZICARI, R. (Eds.). **Conceptual modeling, databases, and case: An integrated view of information systems development.** New York: John Wiley & Sons, 1992. p. 49–68.

NARDI, J. C.; DE ALMEIDA FALBO, R.; ALMEIDA, J. P. A.; GUIZZARDI, G.; PIRES, L. F.; VAN SINDEREN, M.; GUARINO, N. **Towards a Commitment-Based Reference Ontology for Services.** In: Enterprise Distributed Object Computing Conference (EDOC). In Proceedings... Vancouver, Canada: IEEE, 2013. P. 175–184.

NARDI, J. C.; FALBO, R. DE A.; ALMEIDA, J. P. A. **Foundational Ontologies for Semantic Integration in EAI: A Systematic Literature Review** (C. Douligeris, N. Polemi, A. Karantijas, W. Lamersdorf, Eds.). In: IFIP Conference on e-Business, e-Services, e-Society (I3E). In Proceedings... Athens, Greece: Springer Berlin Heidelberg, 2013. P. 238–249.

OLIVÉ, A. **Conceptual Modeling of Information Systems.** Berlin, Germany: Springer-Verlag, 2007. p. 455

OMG. **Semantic Information Modeling for Federation (SIMF): Request for Proposal.** Needham, USA: 2011a. 45 p.

OMG. **OMG Unified Modeling Language (OMG UML), Infrastructure - Version 2.4.1.** 2011b. 877 p.

PEREIRA, D. C.; ALMEIDA, J. P. A. **Representing Organizational Structures in an Enterprise Architecture Language.** In: Workshop on Formal Ontologies meet Industry (FOMI). In Proceedings... Rio de Janeiro, Brazil: IOS Press, 2014.

PERGL, R.; SALES, T. P.; RYBOLA, Z. **Towards OntoUML for Software Engineering: From Domain Ontology to Implementation Model.** In: International

Conference on Model & Data Engineering (MEDI). In **Proceedings...** Amantea, Italy: Springer Berlin Heidelberg, 2013. P. 249–263.

POVEDA, M.; SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A. **Common Pitfalls in Ontology Development**. In: Current Topics in Artificial Intelligence, CAEPIA 2009 Selected Papers. In **Proceedings...** 2010a. P. 91–100.

POVEDA, M.; SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A. **A Double Classification of Common Pitfalls in Ontologies**. In: Workshop on Ontology Quality (OntoQual). In **Proceedings...** Lisbon, Portugal: CEUR-WS.org, 2010b. P. 1–12.

POVEDA-VILLALÓN, M.; SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A. **Validating Ontologies with OOPS!** In: International Conference on Knowledge Engineering and Knowledge Management (EKAW). In **Proceedings...** Galway City, Ireland: Springer Berlin Heidelberg, 2012. P. 267–281.

QUERALT, A.; TENIENTE, E. Verification and Validation of UML Conceptual Schemas with OCL Constraints. **ACM Transactions on Software Engineering Methodology**, v. 21, n. 2, p. 13, 2012.

RECKER, J.; ROSEMANN, M.; GREEN, P. F.; INDULSKA, M. Do Ontological Deficiencies in Modeling Grammars Matter? **MIS Quarterly**, v. 35, n. 1, p. 57–79, 2011.

ROUSSEY, C.; ZAMAZAL, O. **Antipattern detection: how to debug an ontology without a reasoner** (P. Lambrix, G. Qi, M. Horridge, B. Parsia, Eds.). In: International Workshop on Debugging Ontologies and Ontology Mappings (WoDOOM). In **Proceedings...** : CEUR Workshop Proceedings. Montpellier, France: CEUR-WS.org, 2013. P. 45–56.

SALES, T. P. **Identificação de Padrões de Erro em Modelagem Conceitual Por Meio de Validação de Ontologias OntoUML Utilizando Alloy**. Vitória, Brazil: Universidade Federal do Espírito Santo, 2012.

SALES, T. P.; BARCELOS, P. P. F.; GUIZZARDI, G. **Identification of Semantic Anti-Patterns in Ontology-Driven Conceptual Modeling via Visual Simulation**. In: International Workshop on Ontology-Driven Information Systems (ODISE). In **Proceedings...** Graz, Austria: 2012.

SALES, T. P.; GUIZZARDI, G. **Detection, Simulation and Elimination of Semantic Anti-patterns in Ontology-Driven Conceptual Models**. In: International Conference on Conceptual Modeling (ER). In **Proceedings...** Atlanta, USA: Springer Berlin Heidelberg, 2014.

SCHMIDT, G.; STROHLEIN, T. **Relations and Graphs: Discrete Mathematics for Computer Scientists**. 1st. ed. Berlin, Germany: Springer, 1993. p. 301

SOBRAL, V.; GUERSON, J. O.; SALES, T. P. **OntoUML MDG Technology v1.2 - Sparx Enterprise Architect User Guide**. Vitória, Brazil: Ontology and Conceptual

Modeling Research Group (NEMO), 2012. 23 p. Available at:
<https://code.google.com/p/ontouml-lightweight-editor/>

STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. **EMF Eclipse Modeling Framework**. 2nd. ed. Boston, USA: Addison-Wesley Professional, 2008. p. 722

SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A.; FERNÁNDEZ-LÓPEZ, M. The NeOn Methodology for Ontology Engineering. In: SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A.; MOTTA, E.; GANGEMI, A. (Eds.). **Ontology Engineering in a Networked World**. Berlin, Germany: Springer Berlin Heidelberg, 2012. p. 444.

SURE, Y.; GÓMEZ-PÉREZ, A.; DAELEMANS, W.; REINBERGER, M.-L.; GUARINO, N.; NOY, N. F. Why Evaluate Ontology Technologies? Because It Works! **IEEE Intelligent Systems**, v. 19, n. 4, p. 74–81, 2004.

TAHWIL, M. F. **An AntiPattern-Based OWL Ontology Debugging Tool**. Madrid, Spain: Universidad Politécnica de Madrid, 2010.

TORT, A.; OLIVÉ, A. An approach to testing conceptual schemas. **Data & Knowledge Engineering**, v. 69, n. 6, p. 598–618, jun. 2010.

TORT, A.; OLIVÉ, A.; SANCHO, M.-R. An approach to test-driven development of conceptual schemas. **Data & Knowledge Engineering**, v. 70, n. 12, p. 1088–1111, dez. 2011.

VVRANDEČIĆ, D. Ontology Evaluation. In: STAAB, S.; STUDER, R. (Eds.). **Handbook on Ontologies**. International Handbooks on Information Systems. 2nd. ed. Berlin: Springer Berlin Heidelberg, 2009. p. 293–313.

VVRANDEČIĆ, D. **Ontology Validation**. Karlsruhe, Germany: Karlsruher Instituts für Technologie (KIT), 2010.

W3C. **OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax**. 2012.

W3C. **RDF 1.1 Concepts and Abstract Syntax**. 2014.

ZAMBORLINI, V.; GUIZZARDI, G. **On the Representation of Temporally Changing Information in OWL**. In: International Workshop on Vocabularies, Ontologies and Rules for The Enterprise (VORTE). In **Proceedings...** Vitória, Brazil: IEEE Computer Society, 2010. P. 283–292.

ANNEX A ALLOY

Alloy is a language for modeling structures based on first-order logic and relational calculus. Although undecidable, it allows fully automatic analysis through instance finding. The Alloy Analyzer, a free software tool, enables editing and analyzing Alloy models.

In this annex, we briefly describe the Alloy language, explaining the operators that compose its logic, the syntactical constructs that are relevant for our work and how one can analyze Alloy models. The content of the following subsections is mostly extracted from (JACKSON, 2012), so quotation marks and repeated references will be omitted.

A.1 LOGIC

We start the description with the logic that support Alloy, a combination of first order logic quantifiers with relational calculus, baptized as *relational logic*.

Alloy builds everything upon two abstractions: atoms and relations.

Atoms are the most primitive structure. They are indivisible, for atoms have no parts. They are also immutable, since their properties never change. Lastly, they are uninterpreted, meaning that they do not have any built-in properties.

Relations are structures that relate atoms. They can be understood as a set of tuples of any size, each tuple being an ordered sequence of atoms. The easy analogy is to a database table. Each tuple corresponds to a row, and each entry of the tuple is a column. The number of tuples in a relation (or rows in a table) is its size. Any size is possible, including zero. Moreover, the number of entries in the tuple (or the number of columns in a table) is its arity, and must always be greater than one.

Alloy represents scalars as unary relations with one tuple (singletons). Sets are relations with one column, but any number of rows. Relations with arity greater or equal to two capture mappings between atoms (we commonly refer to them as relations). To clarify, we provide an example for scalars, sets and binary relations:

- scalar: myName = {(James)}
- set: Person = {(Luke), (Joseph), (James), (John)}
- binary relation: friendOf {(James, Luke), (Luke, James), (John, Joseph)}

The **domain** of a relation is the set of atoms in its first column; the **range** is the set in the last column. In our previous example, the domain of friendOf is {(James), (Luke), (John)} and the range is {(Luke), (James), (Joseph)}.

Moreover, alloy defines three **constants**: *univ*, correspond to the set of all existing atoms; *none* represents the empty set; and *iden* stands for the identity binary relation, i.e., a relation of every atom to itself. In our previous example *univ* would be the set {(Luke), (Joseph), (James), (John)}, whilst *iden* would be equal to {(James, James), (Luke, Luke), (John, John), (Joseph, Joseph)}.

Furthermore, Alloy defines 4 categories of operators: set, relational, logical and quantification. Set operators are binary and apply to all relations, regardless of their internal structure (unary, binary, etc.). Alloy has all the basic set-theory operations: union (+), intersections (&), difference (-), subsetting (in) and equality (=)

Alloy's relational operators are more complex and differently from the set one, the internal structure of a relation matters. They are: arrow/product (->), dot join (.), box join ([]), transpose (~), transitive closure (^), domain restriction (<:) and range restriction (:>).

The **arrow** operator, also referred to as product, corresponds to the Cartesian product of the two relations. It results in a new relation, whose arity is the sum of the previous arities. For example, {(A)} -> {(B), (C)} = {(A,B), (A,C)}.

The join operators, both **dot** and **box** are very similar to the relational database join or even OCL's dot operator. The dot join matches the last atom of the first tuple to the first atom of the second tuple, if they are the same, a resultant tuple is created, composed by the first atoms of the first tuple, followed by the second atoms of the second tuple. For example, {(A, B), (C, D)} . {(B,E), (B,F), (D,G)} = {(A, E), (A, F), (C, G)}. The box join has the same effect as the dot, but it just takes the arguments in different orders: $e1[e2]$ is equivalent to $e2.e1$.

The **transpose** operation is a unary operation that is only applicable to binary relations. It results in another binary relation that is the mirror image of the original one, i.e., contains the same amount of tuples but all of them in the reversed order. For example, the transposition of friendOf is $\{(Luke, James), (James, Luke), (Joseph, John)\}$. This operator is useful to define symmetric relations, as in `rel==~rel` and also inverse relations, `fatherOf==~hasFather`

The **transitive closure** is another unary relation that is only applicable for binary relations. It can be computed by taking the inputted relation, adding the join of the relation with itself, then adding the join of the relation with that, and so on: ${}^n r = r + r.r + r.r.r + \dots$

The **restriction** operators are used to filter relations to a given domain or range. They take as parameters a set and a relation of any arity. A domain restriction expressed as $s <: r$ (s being a set s and r a relation r) results in a relation with the same arity of r but containing only tuples that start with atoms contained in s . For example, $\{(Luke), (James)\} :> \{(Luke, James), (Joseph, James), (James, John), (Luke, Joseph)\} = \{(Luke, James), (James, John), (Luke, Joseph)\}$. Range restrictions are analogous, but the analysis is on the relations range (the last atom in the tuple).

Alloy uses logical operators contained in standard propositional logic, providing for each a shorthand and a verbose representation: negation (!, not), conjunction (&&, and), disjunction (||, or), implication ($=>$, implies) and bi-implication ($<=>$, iff).

Moreover, Alloys uses quantification operators defined in predicate calculus, which aid users in constraint specification. The possibilities are:

- universal quantification, represented as `all x: e | F` and meaning that F holds for every x in e ;
- existential quantification, represented as `some x: e | F` and meaning that F holds for at least one x in e ;
- existential negation, represented as `no x: e | F` and implying that F holds for no x in e ;
- `one x: e | F`, which means that F holds for at most one x in e ;
- `sig x: e | F`, representing a constraint F that holds for exactly one x in e .

Finally, users can use the set size operator (#) in a relation with any arity to obtain the number of tuples it contains, as an integer value. For example, #{{(A),(B),(C),(D)}}=4 and #{{(A,B)}}=1

A.2 SYNTAX

Alloy is a small language. A generic specification usually contains a module header, signatures declarations, constrain paragraphs, assertions and commands. We use the example shown in Listing 24 to illustrate how to combine Alloy's constructs to produce a valid specification. The example formalizes a domain that contains the general concepts of Person, refined in Man and Woman, and Animal, refined in Car and Dog. Moreover, we formalize paternal and maternal relations amongst people and the pet ownership between people and animals.

Listing 24. An Alloy specification.

```

1  module example/listing1
2  open util/relation
3
4  sig Person {
5      pet: set Pet,
6      father: lone Man,
7      mother: lone Woman
8  }
9  sig Man, Woman in Person { }
10
11 abstract sig Animal {}
12 {
13     #(pet.this)<=1
14 }
15 sig Cat, Dog extends Pet { }
16
17 fact gender {
18     Person = Man + Woman
19     disj [Man, Woman]
20 }
21
22 fact {
23     all x : Person | x not in x.^{father + mother}
24 }
25
26 fun parents (p: Person): set Person{
27     p.father + p.mother
28 }
29

```

```

30  pred atLeastOneParent{
31      some x: Person | some parents[x]
32  }
33  run atLeastOneParent for 5 but 2 Woman
34
35  assert noSelfParent {
36      irreflexive[father+mother]
37  }
38  check noSelfParent for 3
39

```

An Alloy specification can optionally contain a module header. It does have one when users include module declarations and/or module imports.

A **module declaration** starts with the `module` key word, followed by the module name (line 1). It states the name of the specification in the file. Alloy uses the same naming convention used in Java: the full name of the module corresponds to its path and filename in the file system. By default, Alloy modules have the file extension “.als”, which should not be specified in the module name. In our example, the module name is “listing1” and its address in the file system is “*example/listing1.als*”, starting from the working directory of the Alloy Analyzer current running.

A **module import** works just like in other programming languages: imports contents specified in other files and allow reuse. In Alloy, users can reuse predicates, signatures and functions (we explain these concepts in the following paragraphs). The reserved word `open` identifies module importation. In our example, we import the module “util/relation” (line 2), which defines predicates regarding properties of binary relations, such as reflexivity, symmetry, etc.

Furthermore, there are the **signature declarations**, specified using the keyword `sig`. Signatures are declared individually (lines 4 and 11) or in groups (lines 9 and 15). Each signature represents a set of atoms (a unary relation). Our example declares six signatures: Person, Man, Woman, Animal, Cat and Dog.

Signatures can specialize other signatures. They can be subsignatures, if the keyword `in` is used, or extensions, if `extends` is used. Their main difference is that extensions are mutually disjoint, whilst subsignatures are not. In our example, Man and Woman are subsignatures of Person and Dog and Cat are extensions of Animal.

Furthermore, notice that the reserved word `abstract` precedes the Animal signature declaration (line 11). Abstract signatures are the ones that only contain elements that are in their extension signatures, i.e., they are unions of their extensions signatures.

Alloy, unlike some modeling languages, do not restrict multiple inheritance. However, by default, all top-level signatures are mutually disjoint, thus, restricting the specification of multiple inheritance for parent signatures that subset the same top-level signature. In our example, a signature subsetting Animal and Person would always have an empty extension. Conversely, a signature subsetting Man and Woman would be plausible (if it were not for the additional fact enforcing them as disjoint).

Moreover, signatures can own **field declarations**, each introducing a new relation. The domain of the relation is the owner signature and the following expression defines its range. Fields can also contain cardinality constraints. In our example, we declare three fields (lines 5-7), all owned by the Person signature. Their ranges are Pet, Man and Woman, respectively and they all contain cardinality constraints: a person can have zero or many pets, but at most one mother and at most one father.

A specification may optionally contain constraint paragraphs. The keywords `fact`, `fun` and `pred` define the different types of paragraphs, which record different forms of constraints and expressions. There is no restriction regarding the number of constraints paragraphs a model can have, although a model without any seems pointless.

Fact paragraphs contain constraints that must always hold. In our example, we declare two facts. The first, named “gender” (lines 17-20), states that the atoms in the Person signature are the same ones in the union of the signatures Man and Woman. It also states that Man and Woman share now individual, through the `disj` operator. The second fact block (line 22-24) is unnamed and it specifies that people cannot be their own ancestors (using the transitive closure).

If a constraint applies to each element of a signature (universal quantification), one can optionally defines it as a signature fact: a block that immediately follows the signature definition. In our example, Animal contains a signature fact (lines 12-14), which states that every animal is a pet of at most one person.

In Alloy, **functions** allow modularization and reuse. They are very similar to the concept of function in most programming languages: they accept pre-defined parameters and return values of a pre-determined type, according to an expression. They are also useful to improved model readability. We declare a function labeled “parents” in our example (lines 26-28), which take instances of Person as parameters and return a set resulting from the union of one’s father and mother.

Moreover, **predicates** are very similar to functions: they also allow reuse and they can take arguments. The difference is that they store constraints, instead of expressions. The difference between them is that constraints return Boolean values, whilst expressions return relations of any arity and type. To simplify, one may consider a predicate as a function returning a Boolean value. Predicates are useful to declare constraints that will be applied in particular contexts and to provide directions to example generation. In our example, we define one predicate (lines 30-32), which specifies that there is at least one person with at least one parent (a mother or a father).

Furthermore, an alloy model can contain **assertions**. Labeled by the keyword `assert`, they record properties that are expected to hold. When the analyzer checks them, it looks for instantiations to invalidate the contained expressions.

The last type of paragraph in an Alloy model is the **command paragraph**. The keywords `run` and `check` identify command declarations, which instructs the analyzer to perform particular analyses. When running constraints, the analyzer will try to encounter an instance that satisfies them. Conversely, when checking constraints, the analyzer will try to find counter examples that invalidates them.

Assigning a **scope** to commands bounds the investigation space the analyzer will consider. General scope specification provides an upper bound to the number of atoms each top-level signature (the ones that are not subsignature and neither extensions). For example, the run command in our example (line 33) instructs the analyzer to look for instances that satisfy the predicate in a space containing at most five atoms of the Person signature and at most five of the Animal signature. Particular scope definitions assigns bounds to particular top-level or extension signatures (not subsignatures, though). The keyword `but`, after the scope specification, serves that purpose. Line 33 exemplifies its usage,

A.3 ANALYSIS

Running a predicate or checking an assertion can be reduced to the same task: finding some assignment of relations to variables that makes a constraint true.

Alloy's relational logic is undecidable, so it is not possible to build a tool that can automatically check whether an assertion is truly valid or not. Therefore, in a way or another, some compromise is necessary. Formal methods tackle this problem through theorem proving. Though automatic theorem provers exist, they only attempt to construct a proof that an assertion holds. If it succeeds, the assertion is valid. If it fails, however, the assertion may be valid or invalid. Unfortunately, it can be hard to tell whether the failure to verify the assertion was due to a faulty assertion, to limitations of the tool itself, or to a lack of appropriate guidance from the user.

Alloy proposes to compromise in a different way. As a lightweight formal method, it does not try to find proof, but instead look for a counter-proof that invalidates an assertion. It does that by checking the assertion with all possible tests cases within a limit. If the analyzer finds no counterexample, it is still possible that the assertion does not hold, and has a counterexample that is larger than all the considered test cases.

In (JACKSON, 2012), the author argues that instance finding (Alloy's analysis approach), is much more effective than testing, since instead of checking an assertion against a user-defined test base, it exhaustively confronts the assertion to all possibilities within a scope. Furthermore, the usability of an *instance finding* approach is substantiated by the small scope hypothesis, which reads: *Most bugs have small counterexamples*. It means that by analyzing all possibilities for a small scope, one is likely to find a counterexample for an assertion, if one exists. An empirical study has been conducted to evaluate this hypothesis and it show encouraging results (ANDONI et al., 2002).

Alloy is especially useful to detect over and under-constraining in a model. The former refers to a prohibitive model, which does not allow desired instantiations. The latter means an overly permissive model, which allows undesired instantiations. By running predicates, one is able to detect over-constraining and by checking assertion, identify the need for additional constraints.

ANNEX B AUXILIARY ALLOY MODULES

This appendix presents the auxiliary modules required for every transformed model.

The formats used in the Alloy codes are the following: **bolded blue** words are Alloy's reserved words; **light green** indicates comments, preceded by a double dash (--) for single line comments, and a bar followed by star for multi-line comments (*); and **red** for literals.

B.1 SEQUNIV

```

module util/sequniv

open util/integer as ui

/*
 * A sequence utility for modeling sequences as just a
 * relation as opposed to reifying them into sequence
 * atoms like the util/sequence module does.
 *
 * Precondition: each input sequence must range over a prefix
 * of seq/Int.
 *
 * Postcondition: we guarantee the returned sequence
 * also ranges over a prefix of seq/Int.
 *
 * @author Greg Dennis
 */

/** sequence covers a prefix of seq/Int */
pred isSeq[s: Int -> univ] {
    s in seq/Int -> lone univ
    s.ind - ui/next[s.ind] in 0
}

/** returns all the elements in this sequence */
fun elems [s: Int -> univ]: set (Int.s) { seq/Int . s }

/**
 * returns the first element in the sequence
 * (Returns the empty set if the sequence is empty)
 */
fun first [s: Int -> univ]: lone (Int.s) { s[0] }

/**
 * returns the last element in the sequence
 * (Returns the empty set if the sequence is empty)
 */
fun last [s: Int -> univ]: lone (Int.s) { s[lastIdx[s]] }

```

```

/**
 * returns the cdr of the sequence
 * (Returns the empty sequence if the sequence has 1 or fewer element)
 */
fun rest [s: Int -> univ] : s { seq/Int <: ((ui/next).s) }

/** returns all but the last element of the sequence */
fun butlast [s: Int -> univ] : s {
    (seq/Int - lastIdx[s]) <: s
}

/** true if the sequence is empty */
pred isEmpty [s: Int -> univ] { no s }

/** true if this sequence has duplicates */
pred hasDups [s: Int -> univ] { # elems[s] < # inds[s] }

/** returns all the indices occupied by this sequence */
fun inds [s: Int -> univ]: set Int { s.univ }

/**
 * returns last index occupied by this sequence
 * (Returns the empty set if the sequence is empty)
 */
fun lastIdx [s: Int -> univ]: lone Int { ui/max[inds[s]] }

/**
 * returns the index after the last index
 * if this sequence is empty, returns 0
 * if this sequence is full, returns empty set
 */
fun afterLastIdx [s: Int -> univ] : lone Int { ui/min[seq/Int - inds[s]] }

/** returns first index at which given element appears or the empty set if
it doesn't */
fun idxOf [s: Int -> univ, e: univ] : lone Int { ui/min[indsOf[s, e]] }

/** returns last index at which given element appears or the empty set if
it doesn't */
fun lastIdxOf [s: Int -> univ, e: univ] : lone Int { ui/max[indsOf[s, e]] }

/** returns set of indices at which given element appears or the empty set
if it doesn't */
fun indsOf [s: Int -> univ, e: univ] : set Int { s.e }

/**
 * return the result of appending e to the end of s
 * (returns s if s exhausted seq/Int)
 */
fun add [s: Int -> univ, e: univ] : s + (seq/Int->e) {
    setAt[s, afterLastIdx[s], e]
}

/**
 * returns the result of setting the value at index i in sequence to e
 * Precondition: 0 <= i < #s
 */
fun setAt [s: Int -> univ, i: Int, e: univ] : s + (seq/Int->e) {
    s ++ i -> e
}

```

```

/***
 * returns the result of inserting value e at index i
 * (if sequence was full, the original last element will be removed first)
 * Precondition: 0 <= i <= #s
 */
fun insert [s: Int -> univ, i: Int, e: univ] : s + (seq/Int->e) {
    seq/Int <: ((ui/prevs[i] <: s) + (i->e) + ui/prev.((ui/nexts[i] + i) <:
s))
}

/***
 * returns the result of deleting the value at index i
 * Precondition: 0 <= i < #s
 */
fun delete[s: Int -> univ, i: Int] : s {
    (ui/prevs[i] <: s) + (ui/next).((ui/nexts[i] <: s)
}

/***
 * appended is the result of appending s2 to s1
 * (If the resulting sequence is too long, it will be truncated)
 */
fun append [s1, s2: Int -> univ] : s1+s2 {
    let shift = {i', i: seq/Int | int[i'] = ui/add[int[i],
ui/add[int[lastIdx[s1]], 1]] } |
        no s1 => s2 else (s1 + shift.s2)
}

/***
 * returns the subsequence of s between from and to, inclusive
 * Precondition: 0 <= from <= to < #s
 */
fun subseq [s: Int -> univ, from, to: Int] : s {
    let shift = {i', i: seq/Int | int[i'] = ui/sub[int[i], int[from]] } |
        shift.((seq/Int - ui/nexts[to]) <: s)
}

```

B.2 RELATION

```

module util/relation

/*
 * Utilities for some common operations and constraints
 * on binary relations. The keyword 'univ' represents the
 * top-level type, which all other types implicitly extend.
 * Therefore, all the functions and predicates in this model
 * may be applied to binary relations of any type.
 *
 * author: Greg Dennis
 */

/** returns the domain of a binary relation */
fun dom [r: univ->univ] : set (r.univ) { r.univ }

/** returns the range of a binary relation */
fun ran [r: univ->univ] : set (univ.r) { univ.r }

```

```

/** r is total over the domain s */
pred total [r: univ->univ, s: set univ] {
    all x: s | some x.r
}

/** r is a partial function over the domain s */
pred functional [r: univ->univ, s: set univ] {
    all x: s | lone x.r
}

/** r is a total function over the domain s */
pred function [r: univ->univ, s: set univ] {
    all x: s | one x.r
}

/** r is surjective over the codomain s */
pred surjective [r: univ->univ, s: set univ] {
    all x: s | some r.x
}

/** r is injective */
pred injective [r: univ->univ, s: set univ] {
    all x: s | lone r.x
}

/** r is bijective over the codomain s */
pred bijective[r: univ->univ, s: set univ] {
    all x: s | one r.x
}

/** r is a bijection over the domain d and the codomain c */
pred bijection[r: univ->univ, d, c: set univ] {
    function[r, d] && bijective[r, c]
}

/** r is reflexive over the set s */
pred reflexive [r: univ -> univ, s: set univ] {s<:iden in r}

/** r is irreflexive */
pred irreflexive [r: univ -> univ] {no iden & r}

/** r is symmetric */
pred symmetric [r: univ -> univ] {~r in r}

/** r is anti-symmetric */
pred antisymmetric [r: univ -> univ] {~r & r in iden}

/** r is transitive */
pred transitive [r: univ -> univ] {r.r in r}

/** r is acyclic over the set s */
pred acyclic[r: univ->univ, s: set univ] {
    all x: s | x !in x.^r
}

/** r is complete over the set s */
pred complete[r: univ->univ, s: univ] {
    all x,y:s | (x!=y => x->y in (r + ~r))
}

```

```
/** r is a preorder (or a quasi-order) over the set s */
pred preorder [r: univ -> univ, s: set univ] {
    reflexive[r, s]
    transitive[r]
}

/** r is an equivalence relation over the set s */
pred equivalence [r: univ->univ, s: set univ] {
    preorder[r, s]
    symmetric[r]
}

/** r is a partial order over the set s */
pred partialOrder [r: univ -> univ, s: set univ] {
    preorder[r, s]
    antisymmetric[r]
}

/** r is a total order over the set s */
pred totalOrder [r: univ -> univ, s: set univ] {
    partialOrder[r, s]
    complete[r, s]
}
```

B.3 TERNARY

```
module util/ternary

/*
 * Utilities for some common operations and constraints
 * on ternary relations. The keyword 'univ' represents the
 * top-level type, which all other types implicitly extend.
 * Therefore, all the functions and predicates in this model
 * may be applied to ternary relations of any type.
 *
 * author: Greg Dennis
 */

/** returns the domain of a ternary relation */
fun dom [r: univ->univ->univ] : set ((r.univ).univ) { (r.univ).univ }

/** returns the range of a ternary relation */
fun ran [r: univ->univ->univ] : set (univ.(univ.r)) { univ.(univ.r) }

/** returns the "middle range" of a ternary relation */
fun mid [r: univ->univ->univ] : set (univ.(r.univ)) { univ.(r.univ) }

/** returns the first two columns of a ternary relation */
fun select12 [r: univ->univ->univ] : r.univ {
    r.univ
}

/** returns the first and last columns of a ternary relation */
fun select13 [r: univ->univ->univ] : ((r.univ).univ) -> (univ.(univ.r)) {
    {x: (r.univ).univ, z: univ.(univ.r) | some (x.r).z}
}
```

```

/** returns the last two columns of a ternary relation */
fun select23 [r: univ->univ->univ] : univ.r {
    univ.r
}

/** flips the first two columns of a ternary relation */
fun flip12 [r: univ->univ->univ] : (univ.(r.univ))->((r.univ).univ)->(univ.(univ.r)) {
    {x: univ.(r.univ), y: (r.univ).univ, z: univ.(univ.r) | y->x->z in r}
}

/** flips the first and last columns of a ternary relation */
fun flip13 [r: univ->univ->univ] : (univ.(univ.r))->(univ.(r.univ))->((r.univ).univ) {
    {x: univ.(univ.r), y: univ.(r.univ), z: (r.univ).univ | z->y->x in r}
}

/** flips the last two columns of a ternary relation */
fun flip23 [r: univ->univ->univ] : ((r.univ).univ)->(univ.(univ.r))->(univ.(r.univ)) {
    {x: (r.univ).univ, y: univ.(univ.r), z: univ.(r.univ) | x->z->y in r}
}

```

B.4 BOOLEAN

```

module util/boolean

/*
 * Creates a Bool type with two singleton subtypes: True
 * and False. Provides common boolean operations.
 *
 * author: Greg Dennis
 */

abstract sig Bool {}
one sig True, False extends Bool {}

pred isTrue[b: Bool] { b in True }

pred isFalse[b: Bool] { b in False }

fun Not[b: Bool] : Bool {
    Bool - b
}

fun And[b1, b2: Bool] : Bool {
    subset_[b1 + b2, True]
}

fun Or[b1, b2: Bool] : Bool {
    subset_[True, b1 + b2]
}

fun Xor[b1, b2: Bool] : Bool {
    subset_[Bool, b1 + b2]
}

```

```
fun Nand[b1, b2: Bool] : Bool {
    subset_[False, b1 + b2]
}

fun Nor[b1, b2: Bool] : Bool {
    subset_[b1 + b2, False]
}

fun subset_[s1, s2: set Bool] : Bool {
    (s1 in s2) => True else False
}
```

APPENDIX A NOTEWORTHY CONCEPTUAL MODELS

In this appendix, we present and discuss some of the most relevant ontologies we added to our repository.

A.1 THE ITU ONTOLOGIES

The International Telecommunication Union (ITU) is a specialized agency of the United Nations for information and communication technologies. The study groups of ITU's Telecommunication Standardization Sector (ITU-T) develop and maintain international standards known as ITU-T Recommendations. In our repository, we have four ontologies that are formalizations of ITU-T Recommendations, namely The G.805 Ontology, The G.805 Ontology (Revised), The G.800 Ontology and the G.809 Ontology. The ontologies are homonyms to the recommendations they formalize.

We present these four ontologies together in this section because they are a lot similar to one another in terms development context, domain, purpose, etc. In a simple way, these four models describe different aspects and abstraction views of network architectures. Besides G.809, the other three ontologies have been developed and applied in industrial projects between the Electrical Engineering Department of the Universidade Federal do Espírito Santo and a Brazilian telecommunication company. Nonetheless, senior researchers and graduate students have developed them all.

Published in (MONTEIRO et al., 2010), the first version of the ITU-T G.805 the only public available of the four. For that reason, we only provide further details for it.

The G.805 ontology was designed to be a reference model on the domain of transport networks and it has successfully been applied to identify shortcomings on the recommendation it was built upon (BARCELOS et al., 2011). The ontology describes functional and structural technology-independent architectures for transport networks.

G.805 is a reasonably sized ontology, containing 135 classes, 113 associations, 127 generalizations and 36 generalization sets. Note, though, that the ontology does not define any attribute or data type.

Figure 69 details the occurrence of class stereotypes in the G. 805 Ontology. Note that, for this domain, only a subset of OntoUML is required. No class stereotype as Quantity, Collective, Mixin, RoleMixin or Phase was identified. Mostly, this ontology contains relators.

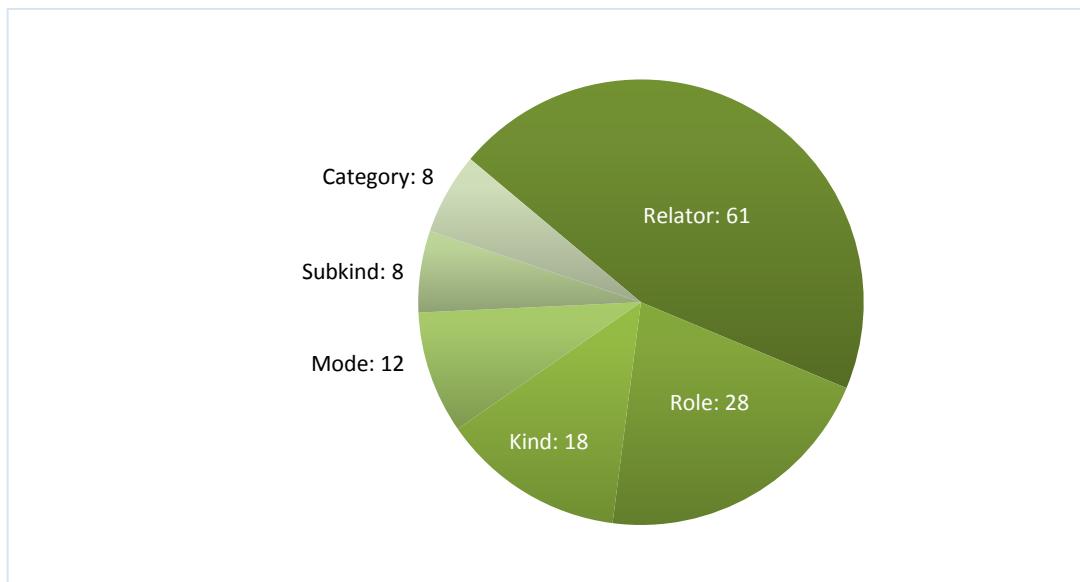


Figure 69. Class' stereotype distribution in the G.805 Ontology.

Finally, Figure 70 presents the distribution of association's stereotypes used in the G.805 ontology. Once more, only a subset of the stereotypes is required: no association is stereotype as SubQuantityOf, SubCollectionOf, MemberOf, Formal or Derivation. We surely expect that, since these relations require quantities, collections and data types, respectively, and there are none in the model.

Another interesting remark regards the comparison between the number of relators and mediations. One of OntoUML's constraints requires every relator to be connected to at least one mediation. However, the number of mediations in G. 805 (38) is close to half the number of relators (61). In fact, this is the lowest mediation/relator rate in the entire repository: 0.62.

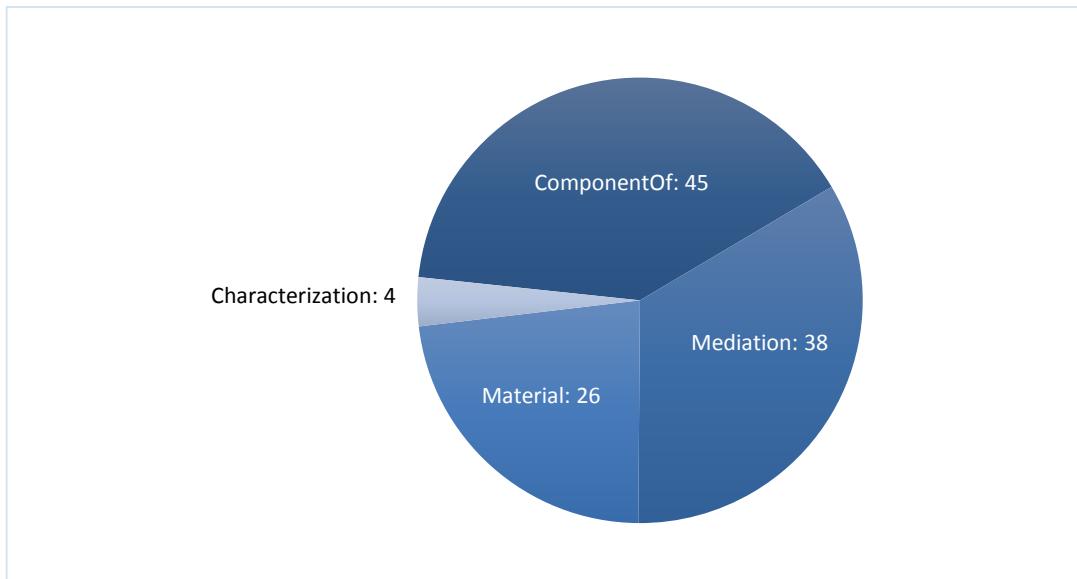


Figure 70. Association's stereotype distribution in the G. 805 Ontology.

A.2 ONTOEMERGEPLAN

The OntoEmerge (FERREIRA, 2013) is an ontology about the domain of emergencies, developed as a main product of a MSc. Thesis in the Universidade Federal do Rio de Janeiro (UFRJ). The ontology aims to support models, systems and strategies associated to the generation, control and support of emergency plans. The authors propose the ontology as a reference to analyze the different elements of a plan, as well as to facilitate the systematic generation of emergency plans.

OntoEmerge builds upon the UFO-C ontology (REF), a model about social interactions. OntoEmerge builds around the concept of emergency plan, i.e. a set of instructions to people should follow in case of a particular type of disaster, like a fire or an earthquake. An emergency plan defines responsibility delegation for carrying out specific actions; identification of personnel, equipment, facilities, supplies, and other resources available for use; and action coordination.

The authors organized the ontology in nine subdomains, namely:

- *Plans, Process and Activities*: describes the composition of emergency plans and their properties
- *Goals, Intentions and Delegations*: focuses on intention of the emergency plans

- *Installation*: describes the facilities for which one creates evacuation plans.
- *Resources, Types of Resources and Roles*: describes the relations between emergency activities and the objects required to perform them, as well as the people who performs them.
- *Human Resources and Materials*: describes the properties and types of objects and agents that participate in emergency activities
- *Environment*: details properties of the environment relevant in case of an emergency.
- *Emergency Event*: characterizes the emergency concept.
- *Risk and Planned Activity*: describe and relate the concepts of Risk, Impact, Hazard, Damage and Vulnerability.
- *Geographical Region*: geo-spatial characterization

In numbers, the ontology is the fourth biggest in our repository. It defines 189 classes, 138 associations, 111 generalizations and 16 generalization sets, distributed in 10 packages.

Figure 71 presents OntoEmerges' class stereotypes distribution. Notice that more than a third of the classes (64 of 189) are defined using unknown stereotypes. That unusual number is justified because the domain formalized by OntoEmerge requires more ontological distinctions than OntoUML currently provides. In order to characterize the domain precisely, meta-concepts like event and higher-order universal are required. In this work, we treat every class qualified by a stereotype not defined in OntoUML as an un-stereotyped class.

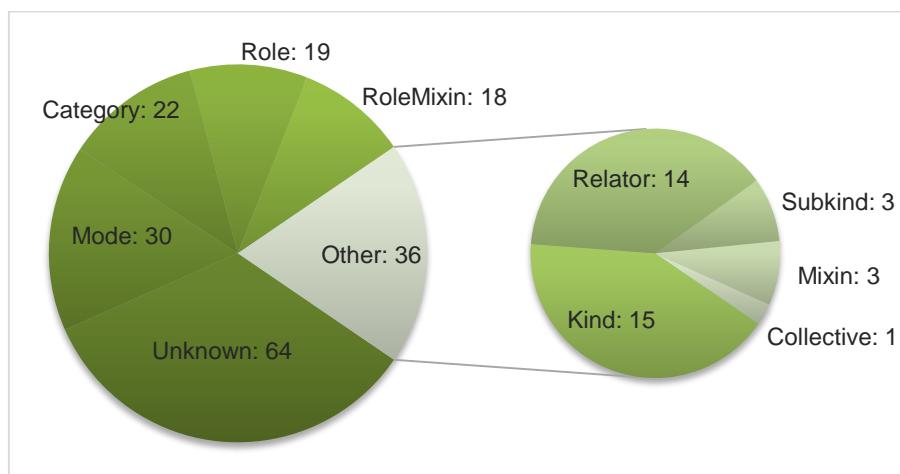


Figure 71. Distribution of OntoEmerge class' stereotypes.

Figure 72 presents the number of occurrences of the association stereotypes in OntoEmerge. More than half of the associations in the ontology are formal (62.3%). This discrepancy with the other relations is due to the high number of unknown stereotypes. Since there is no restriction regarding the use of formal relations, the authors often use it as a last alternative, in this case to relate classes with unknown stereotypes.

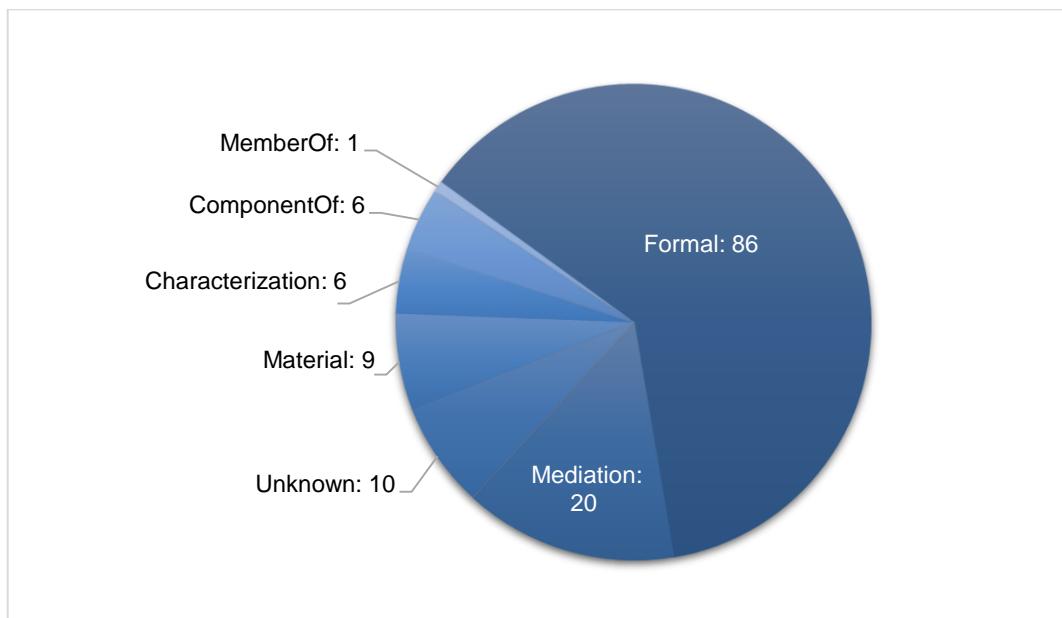


Figure 72. OntoEmerge's association stereotype distribution.

A.3 ONTOBIO

Biodiversity research is natively an interdisciplinary field. The available data comes from all around the world, described and classified by different models and/or standards. The intrinsic heterogeneity of this scenario is aggravated by not only the different necessities and profile of biodiversity experts, but by the huge amount of stored data and the ever-growing number of species.

The OntoBio ontology (ALBUQUERQUE, 2011) proposes to help solve the aforementioned complex interoperability issue of biodiversity research. It was developed in the Brazilian National Center for Amazon Research in collaboration with biodiversity experts.

The authors separate the ontology in five subdomains:

- *Environment*: classifies the types of environment and their properties (e.g. luminosity, climate, soil)
- *Ecosystem*: formalizes the ecosystem's relations and properties, which are relevant for a biodiversity collection protocol.
- *Spatial Location*: describes geo-spatial properties and classifies regions with regard to political, climate and vegetation.
- *Collection*: structures the collection protocol in a high-level of abstraction.
- *Material Entity*: describes the biotic (e.g. plants, animals) and abiotic (e.g. soil and water samples) entities that can be collected.

The ontology's structural numbers are 187 classes, 50 associations, 160 generalizations, 22 generalization sets, 5 data types and 14 attributes. In terms of total elements, it is the fifth largest ontology in the repository. Note that the ontology specifies only 0.27 associations per class, i.e., the ontology defines one association for almost every four classes: the lowest rate encountered in the whole repository. This low association density indicates that the model resembles a lot a taxonomy²¹.

Figure 73 depicts the composition of the OntoBio ontology from the perspective of class stereotype. Notice that the three most frequent stereotypes are Mode, Subkind and Category, in this particular order. Together they correspond to 69.5% of all classes defined in the ontology. The taxonomical nature of this ontology explains this number. For example, there are only five top-level modes within the 52 defined in the ontology.

²¹ We do not use the term "taxonomy" in any sort of judgment, only to help characterize the ontology.

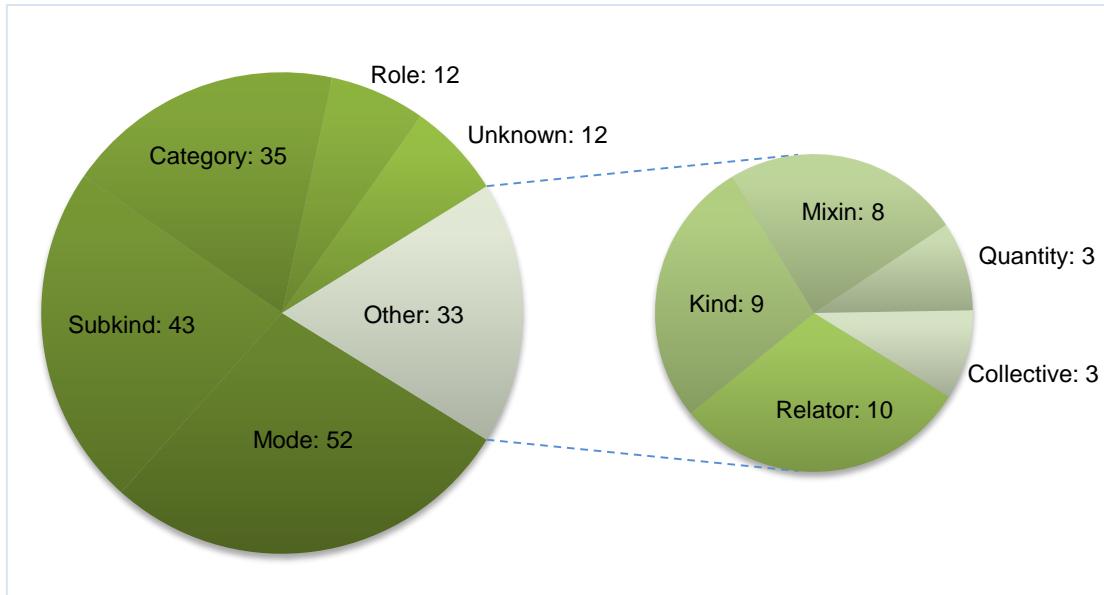


Figure 73. Distribution of the class' stereotypes within the OntoBio ontology.

Lastly, Figure 74 details the occurrence of association stereotypes in the OntoBio ontology. Notice that, just like the OntoEmerge ontology, the Formal associations are also the most frequent. Nine of the fifteen formal relations relate types to their respective high-order universals (their power-types if it were a UML model).

OntoUML does not specify a particular stereotype to relate types and data types. In the OntoBio model, the authors use five associations with unknown stereotypes.

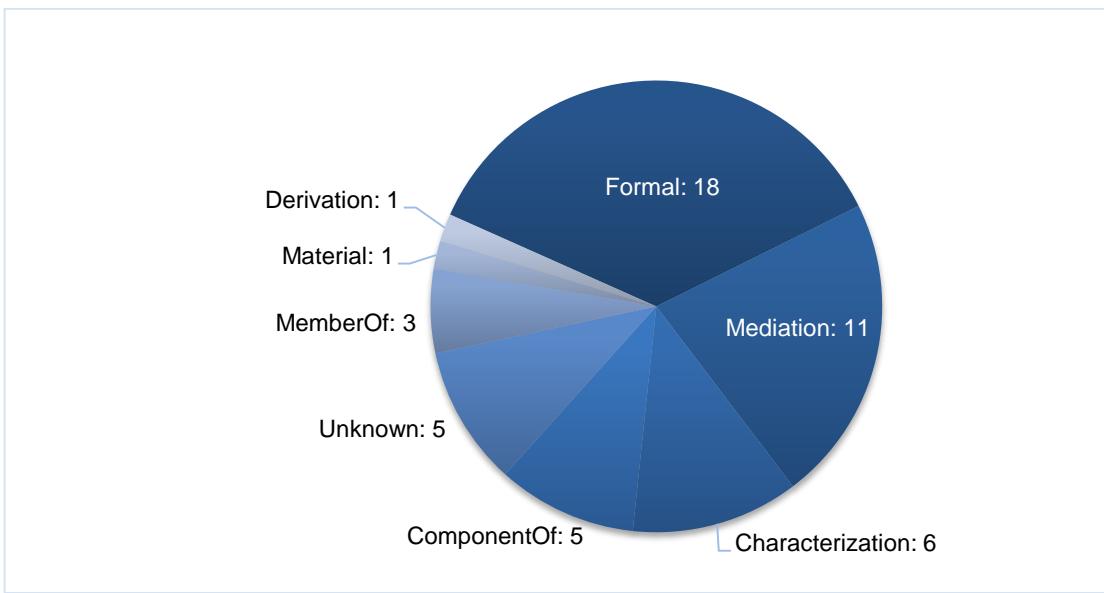


Figure 74. Association's stereotypes distribution in the OntoBio Ontology.

A.4 ONTOUML ORG ONTOLOGY (O3)

The OntoUML Org Ontology (O3) is one of the most recent ontologies we have in our repository. Partially published in (PEREIRA; ALMEIDA, 2014), it is an undergoing masters research being conducted in the Informatics Department of the Universidade Federal do Espírito Santo.

O3 models a subdomain of Enterprise Architecture, named organizational structures or active structure. It concerns who undertakes organizational activities. Therefore, the ontology describes business agents, the tasks they perform, the goals they seek to achieve, alongside with authority relationships, communication lines, work groups, etc. The authors built O3 as a specialization of the UFO-C core ontology, just like OntoEmerge.

The goal of the O3 ontology is to serve as a reference to perform an ontological analysis on the ArchiMate language. As discussed in (PEREIRA; ALMEIDA, 2014), the ontology already allowed the identification of shortcomings in ArchiMate's expressivity and the proposal of more sophisticated modeling constructs for the language.

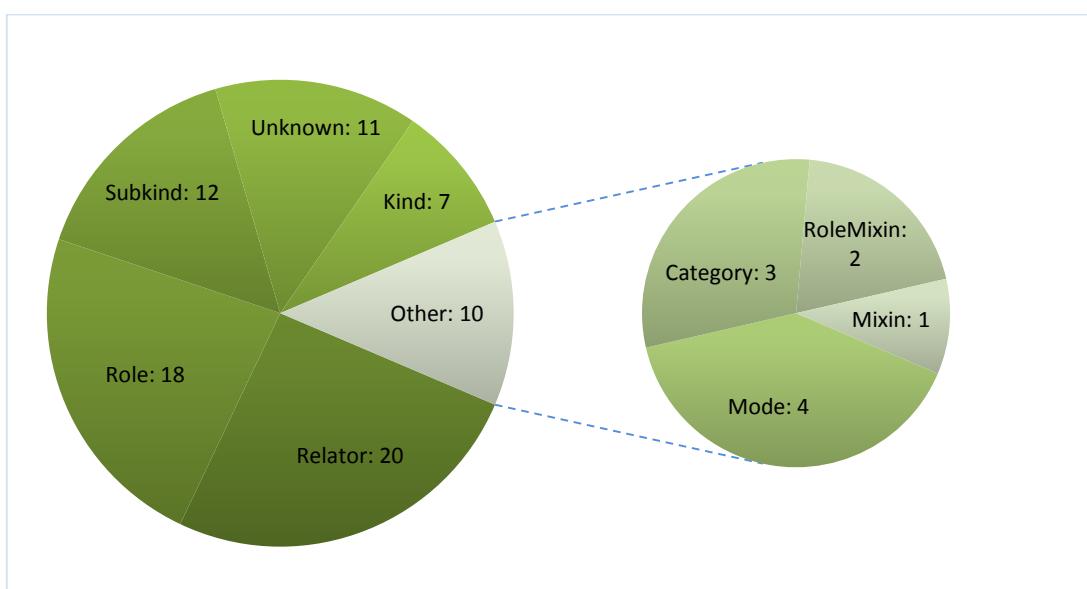


Figure 75. Class distribution in the O3 Ontology.

The model provided by the authors contains 78 classes, 78 associations, 57 generalizations and 8 generalization sets. It contains no data type or attribute. Figure 75 presents O3's class distribution, whilst Figure 76 its association distribution. This

ontology does not define any Quantity, Collective or Phase types and any Derivation, MemberOf, SubCollectionOf or SubQuantityOf of associations.

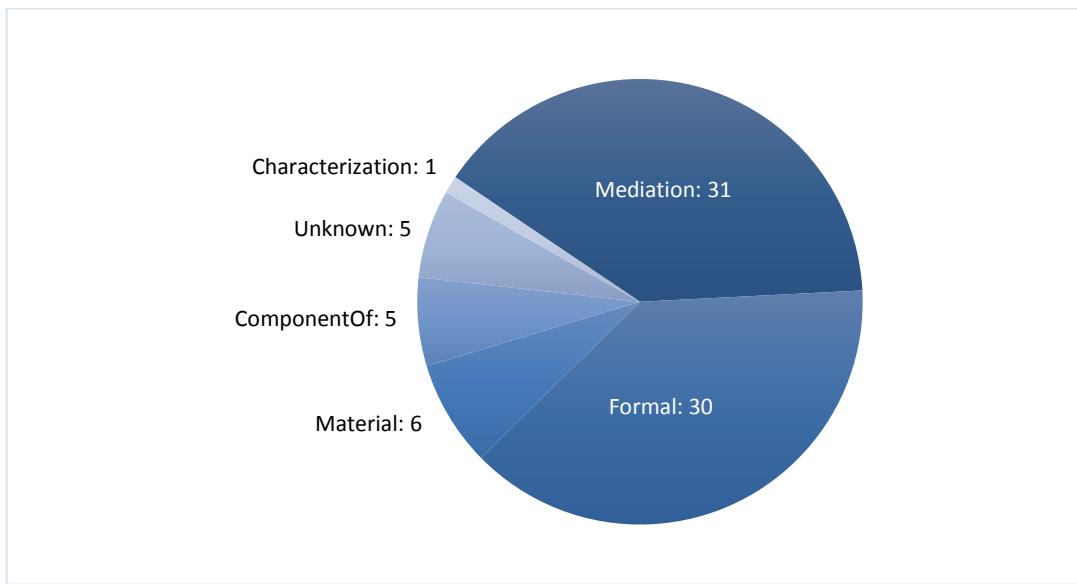


Figure 76. Association's stereotype distribution in the O3 Ontology.

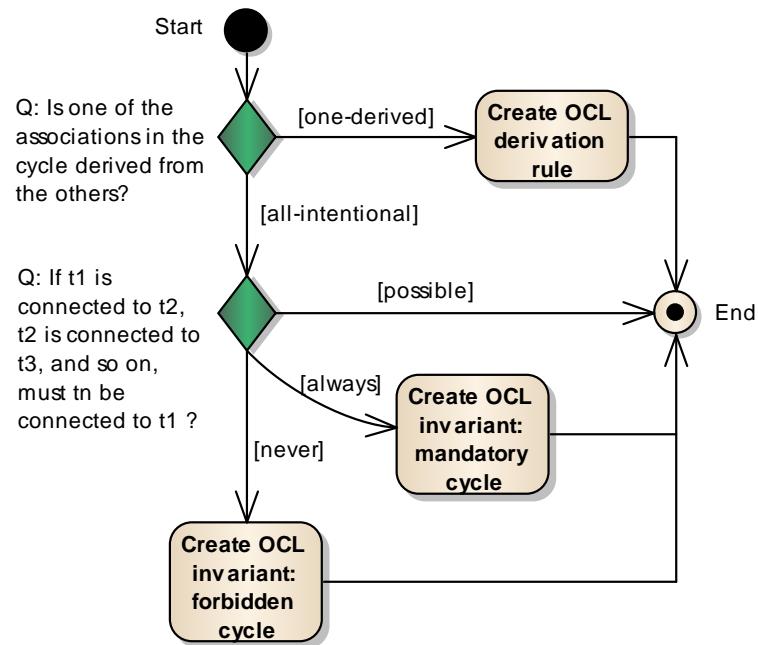
APPENDIX B ANTI-PATTERN ANALYSIS FLOWS

This appendix presents the diagrams of the analysis flows for the anti-patterns presented in Chapter 5. Each diagram contains a fluxogram that details questions and pre-defined possible answers to guide the modeler in deciding whether an occurrence of the anti-pattern indeed is a mistake. If the flow leads to the conclusion that the occurrence is indeed an error, it will also provide the appropriate refactoring solution and describes the steps to perform it.

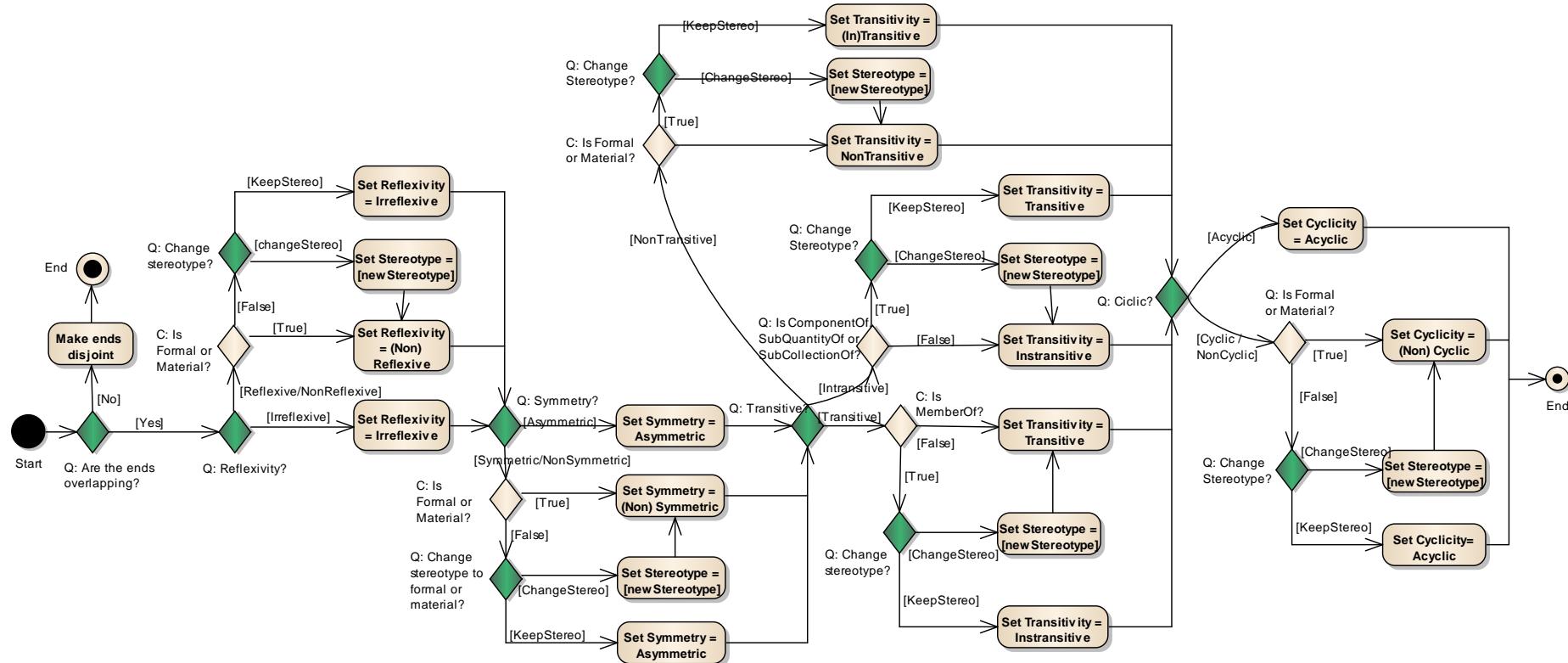
We build the diagrams using basic UML Activity Diagram notation. We adopt a few representation formats to improve readability. Questions that the user must answer are

represented as green diamonds (◆), whilst conditions that require no interaction are represented as yellow diamonds (◇). We represent the possible answers of a question or the possible results of evaluating a condition in brackets (e.g. [yes], [has-identity-provider]). We represent actions to refactor the model by yellow rounded rectangles (Action). The starting point of the analysis is represented by a black circle (●) and the end points by a yellow circle with a concentric black circle (○).

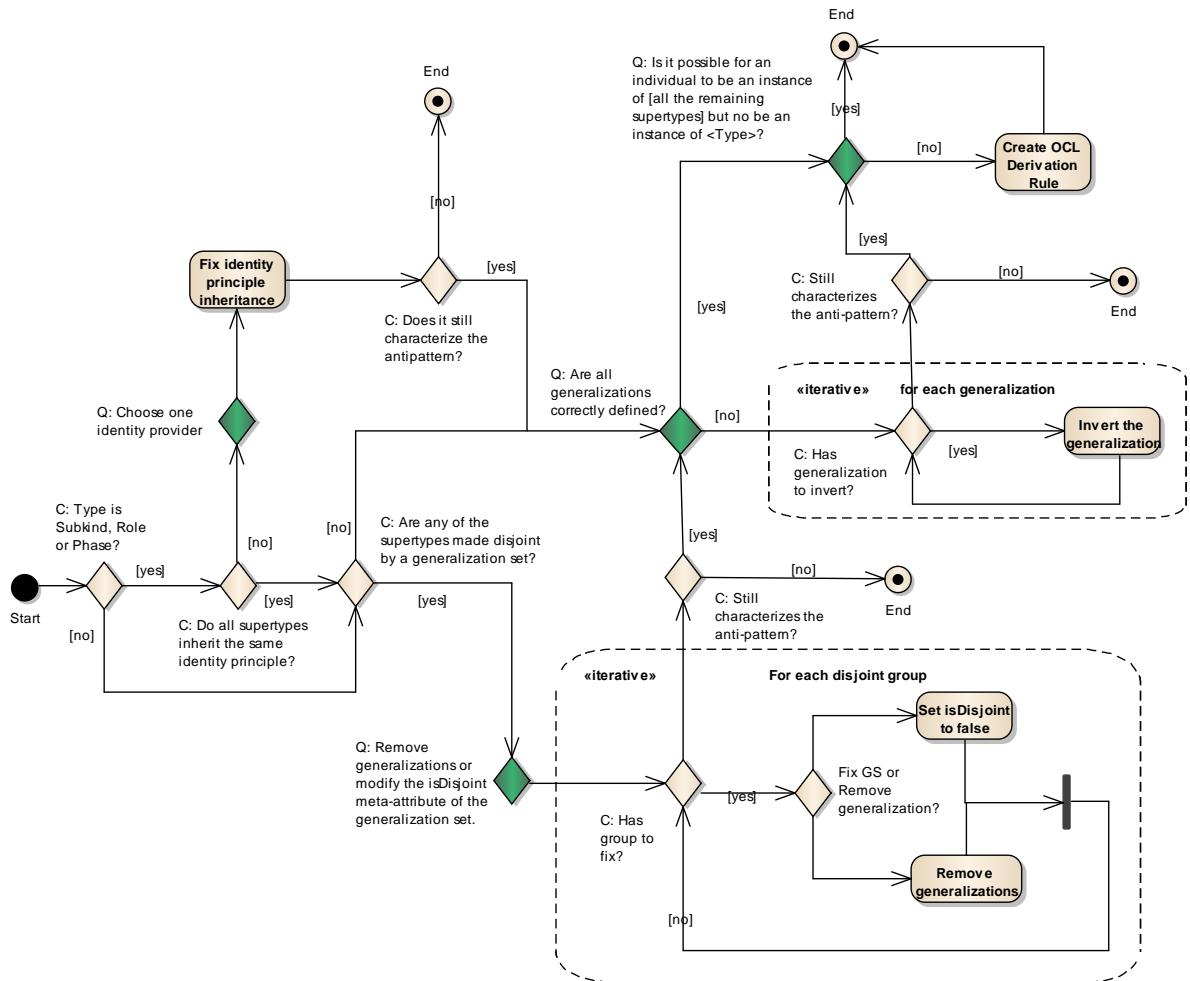
B.1 AssCyc



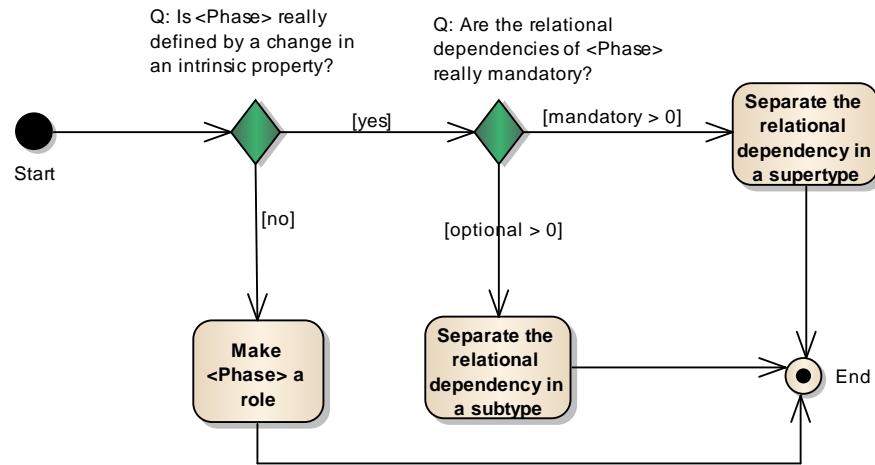
B.2 BINOVER



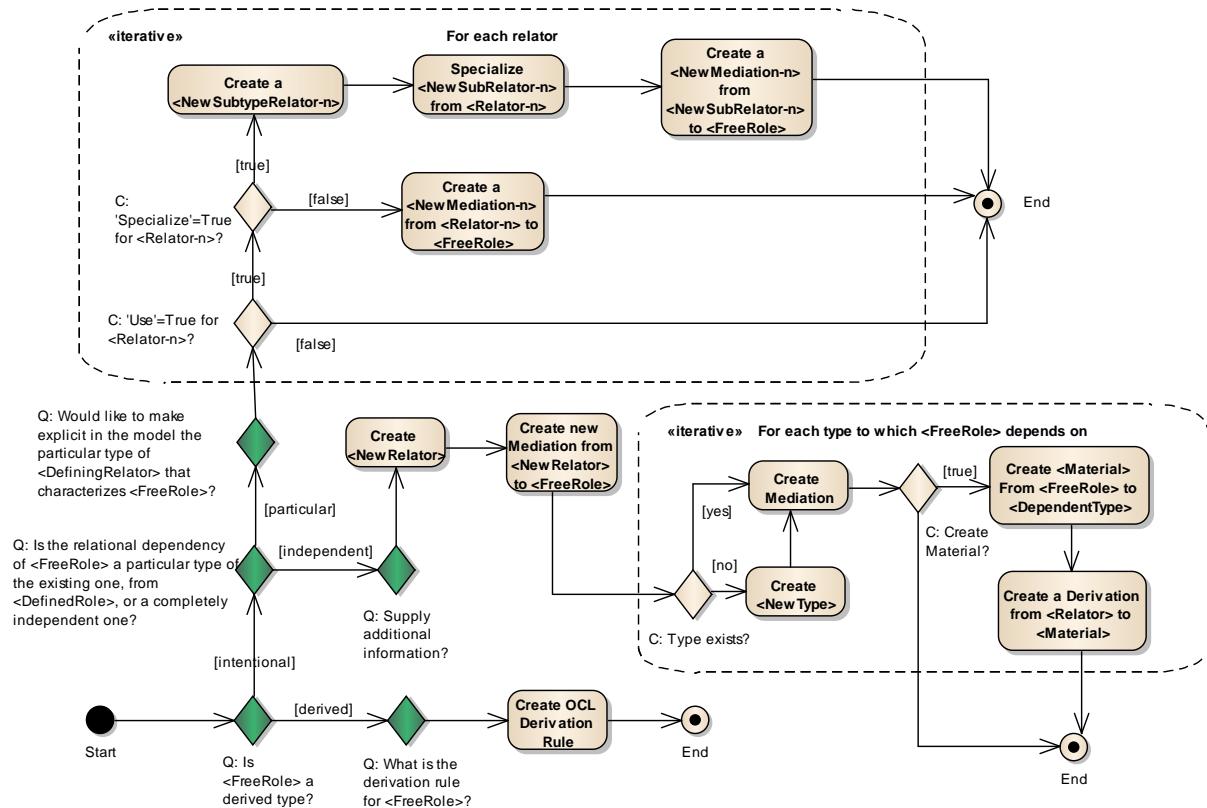
B.3 DECLNT



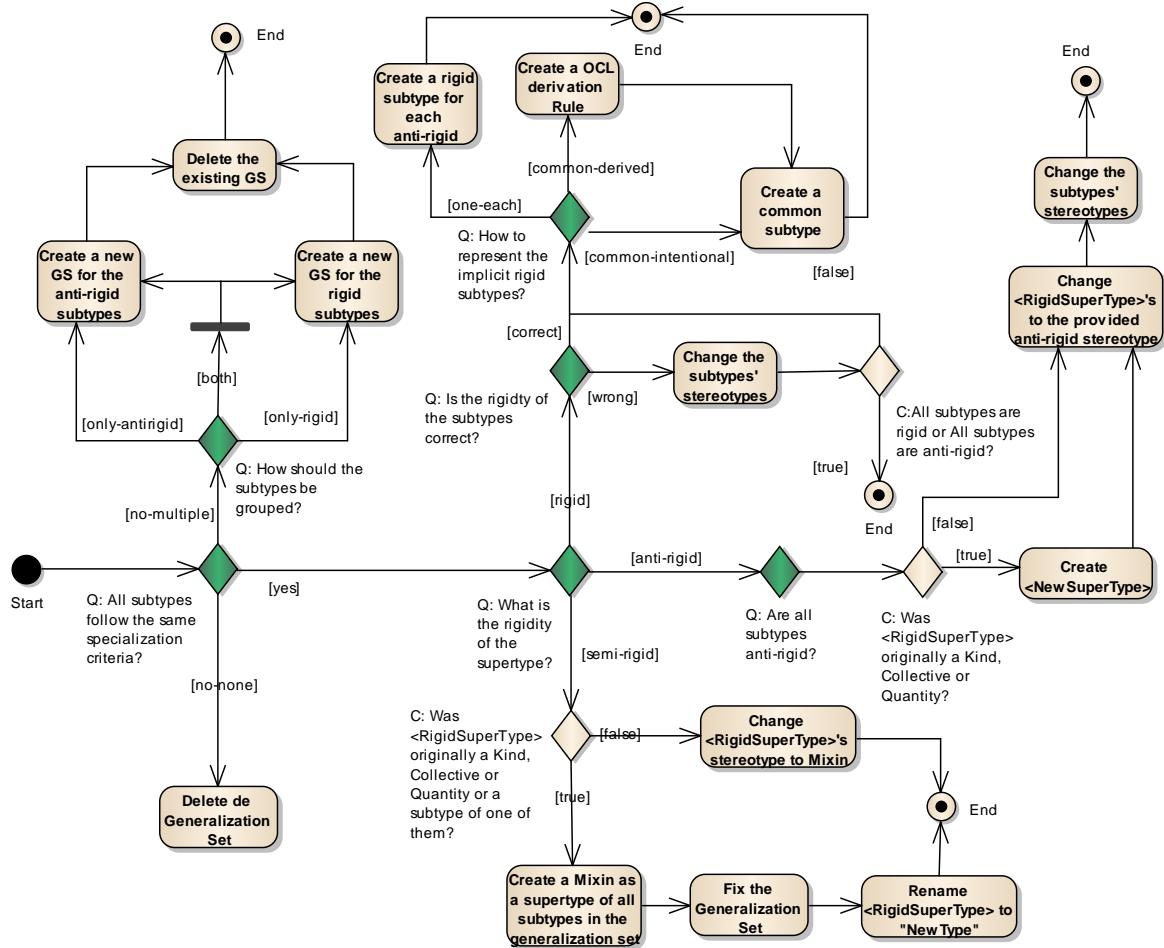
B.4 DEPPhase



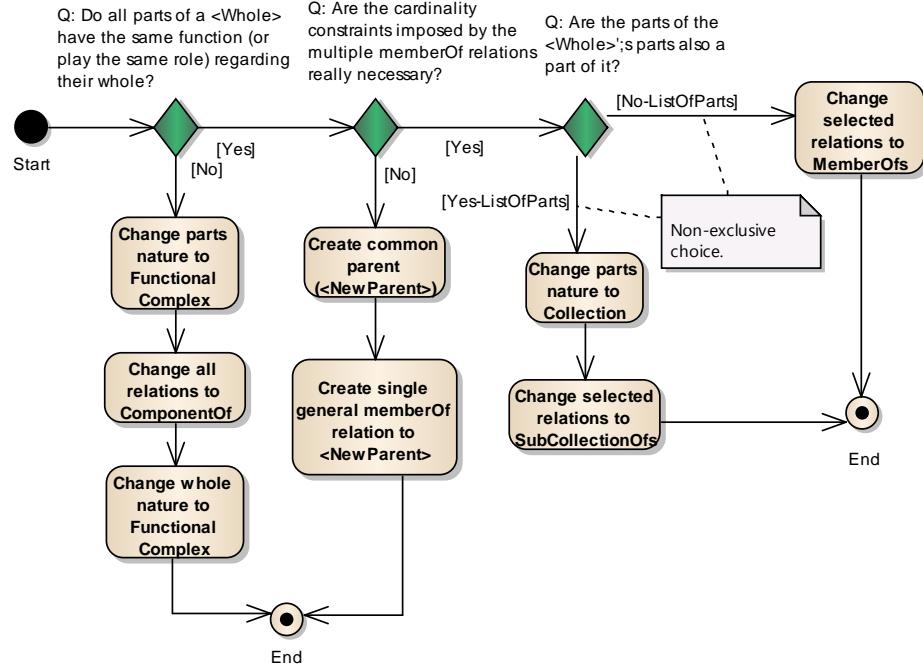
B.5 FREEROLE



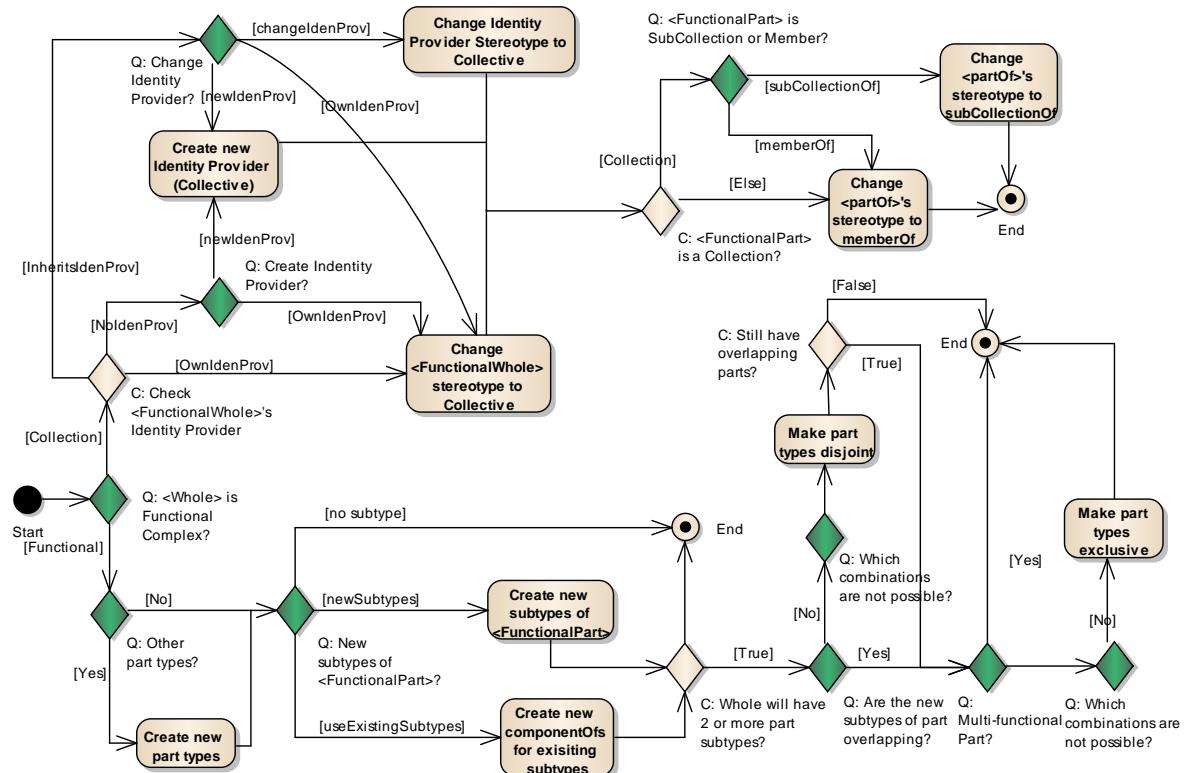
B.6 GSRIG



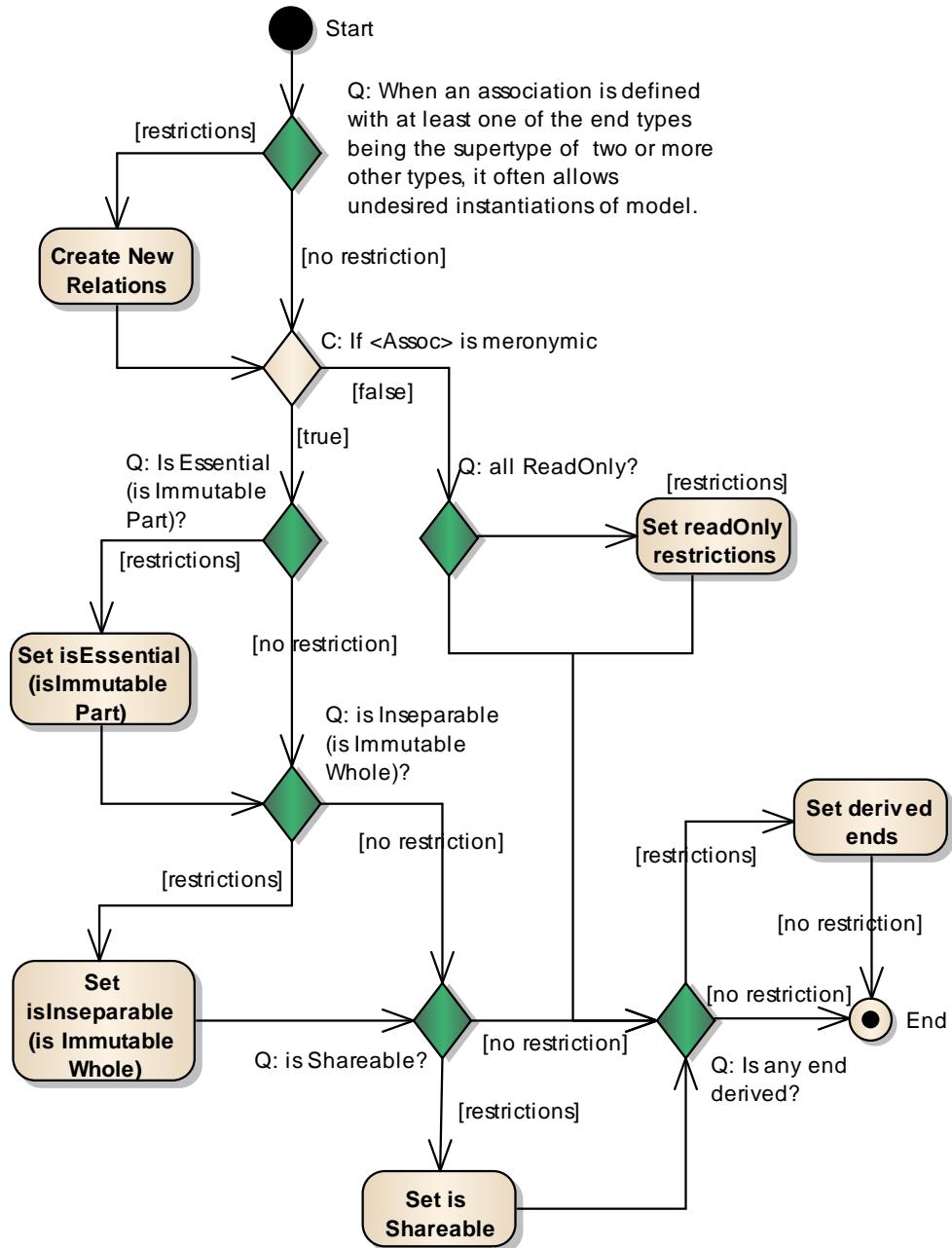
B.7 HETColl



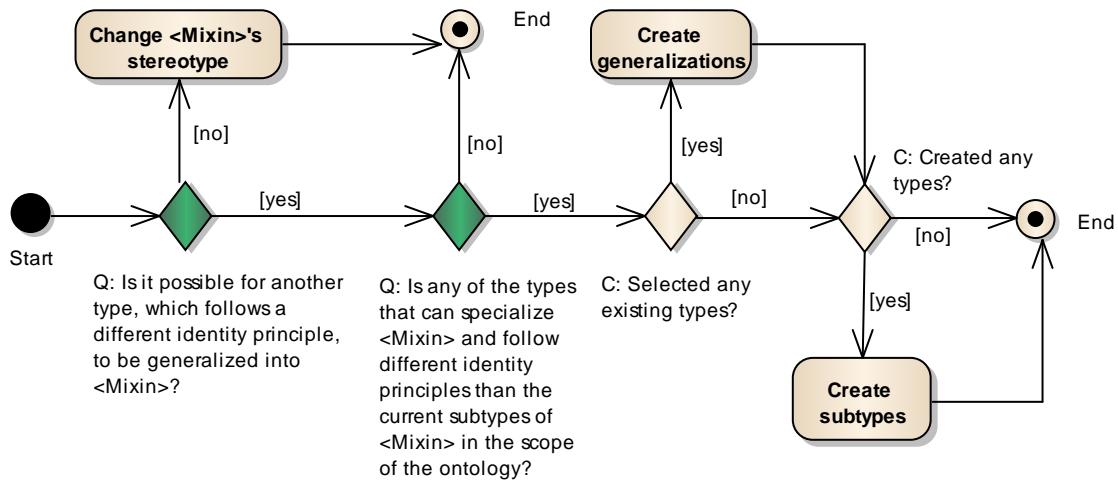
B.8 HomoFUNC



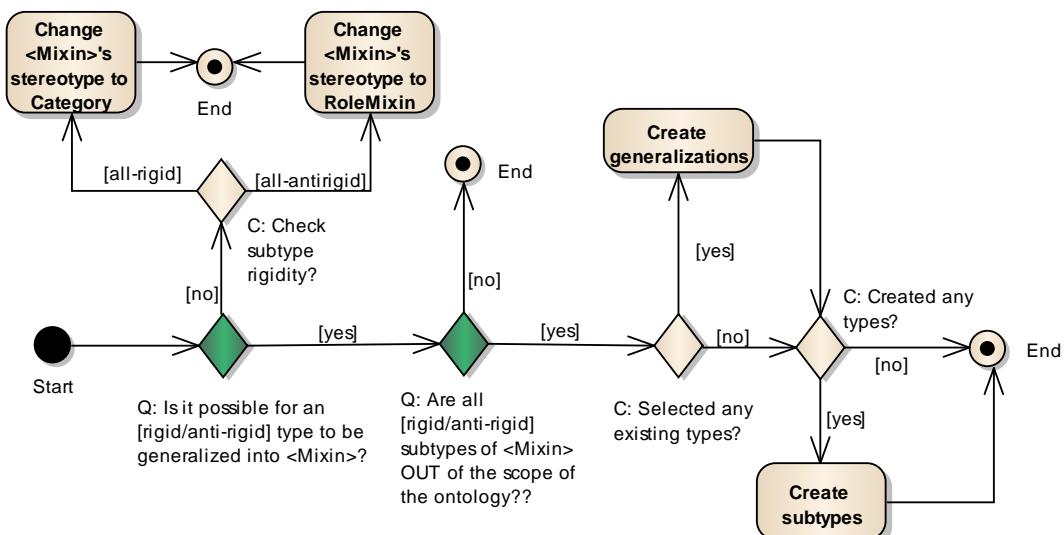
B.9 IMPABS



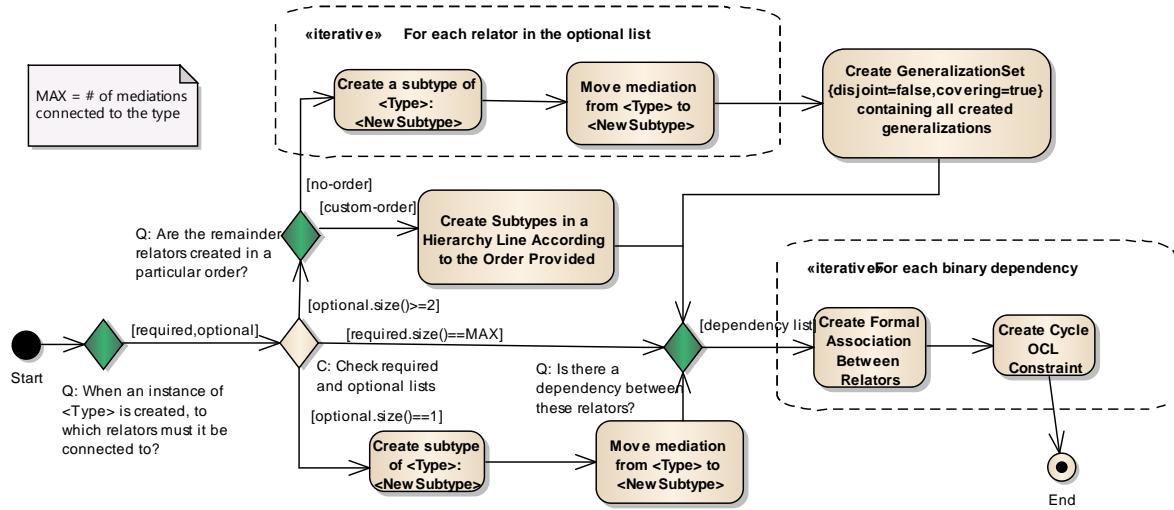
B.10 MIXIDEN



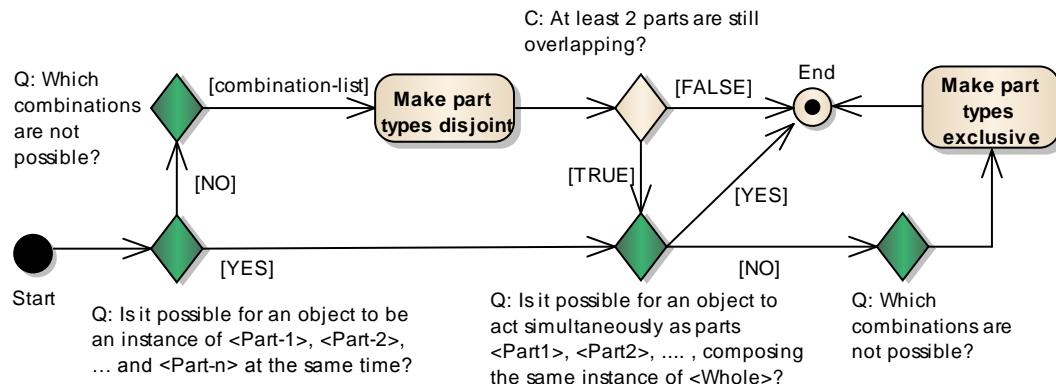
B.11 MixRIG



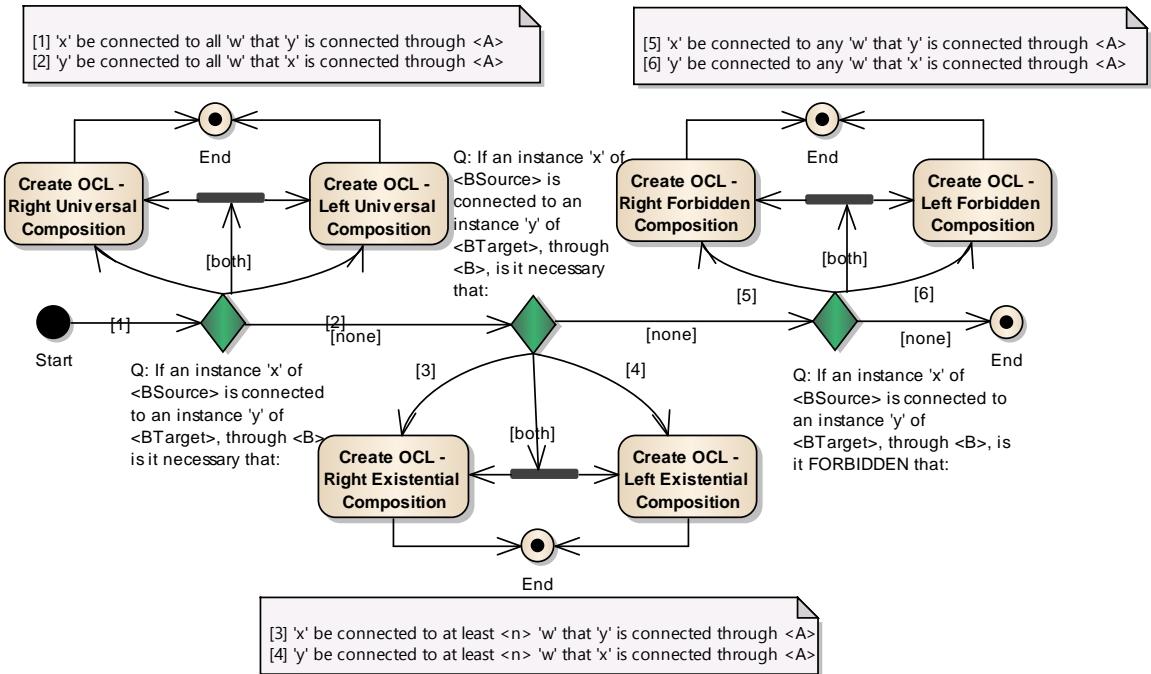
B.12 MULTDEP



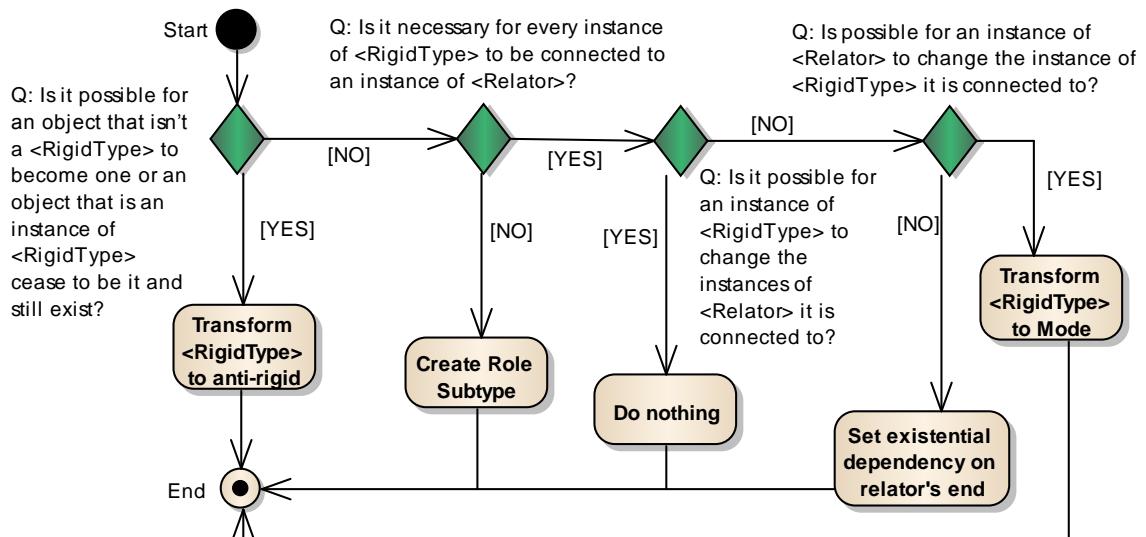
B.13 PARTOVER, RELOVER AND WHOLEOVER



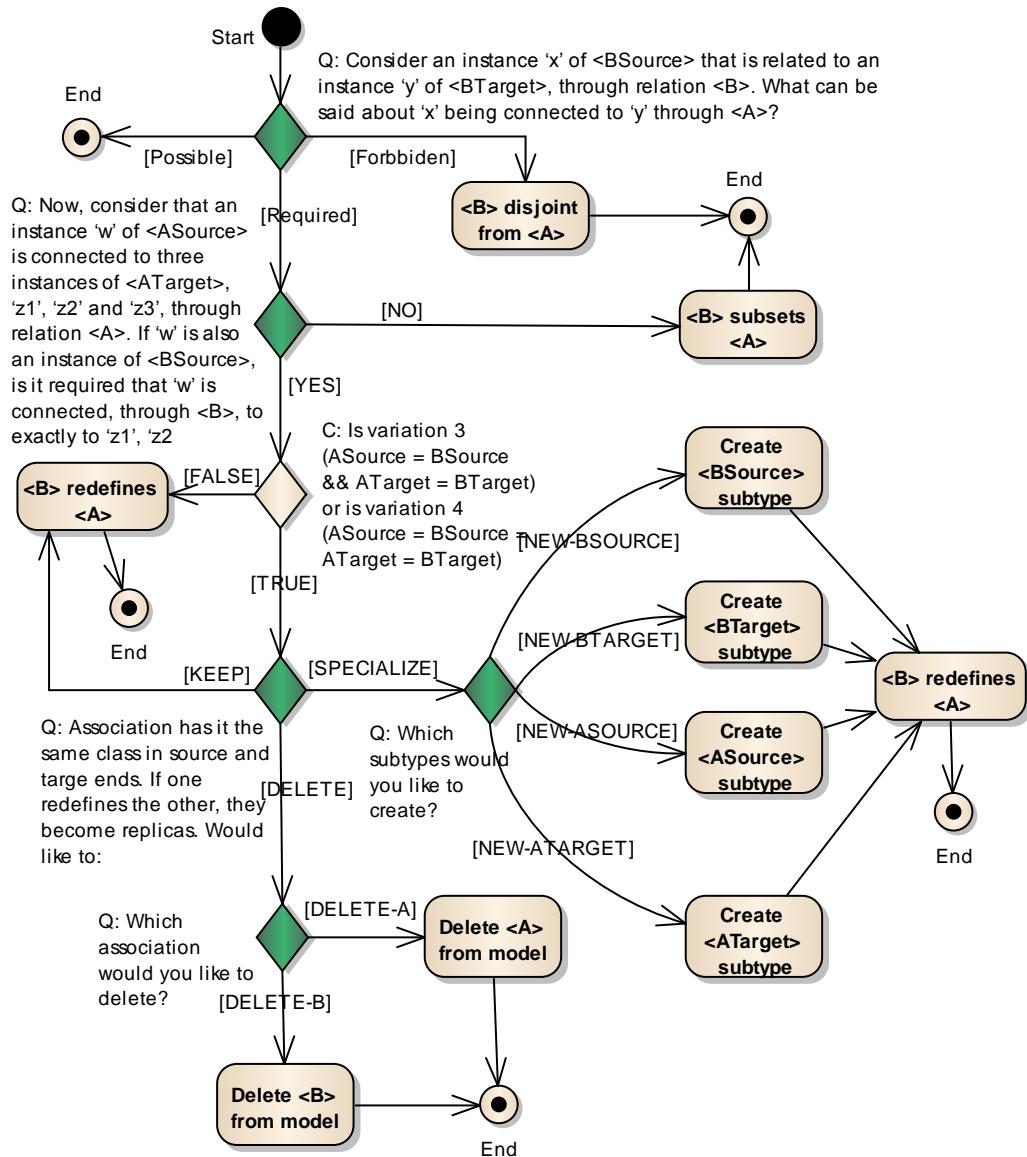
B.14 RELCOMP



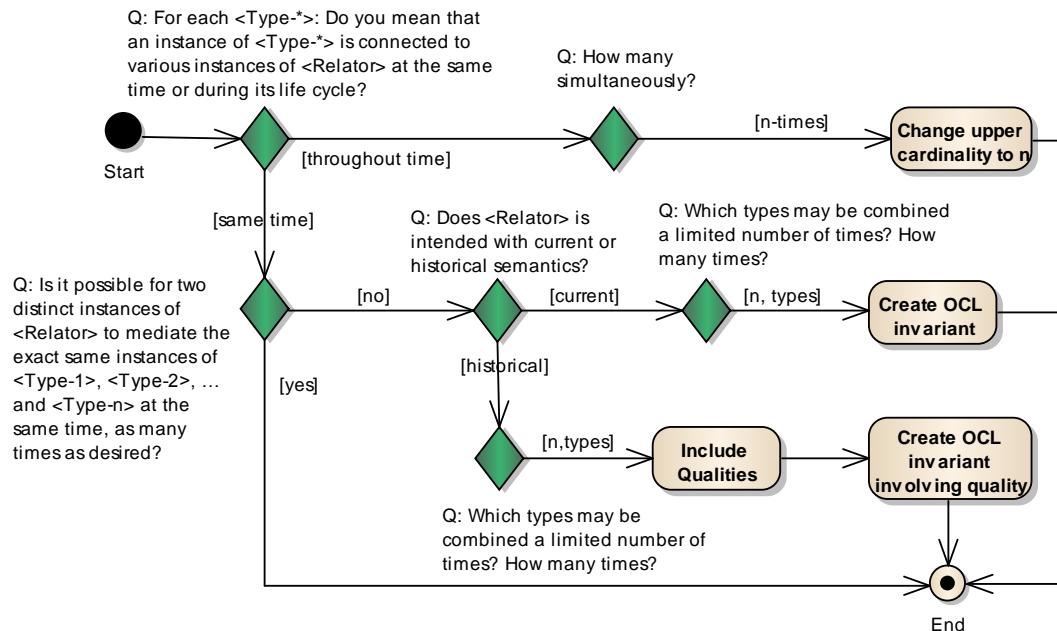
B.15 RELRIG



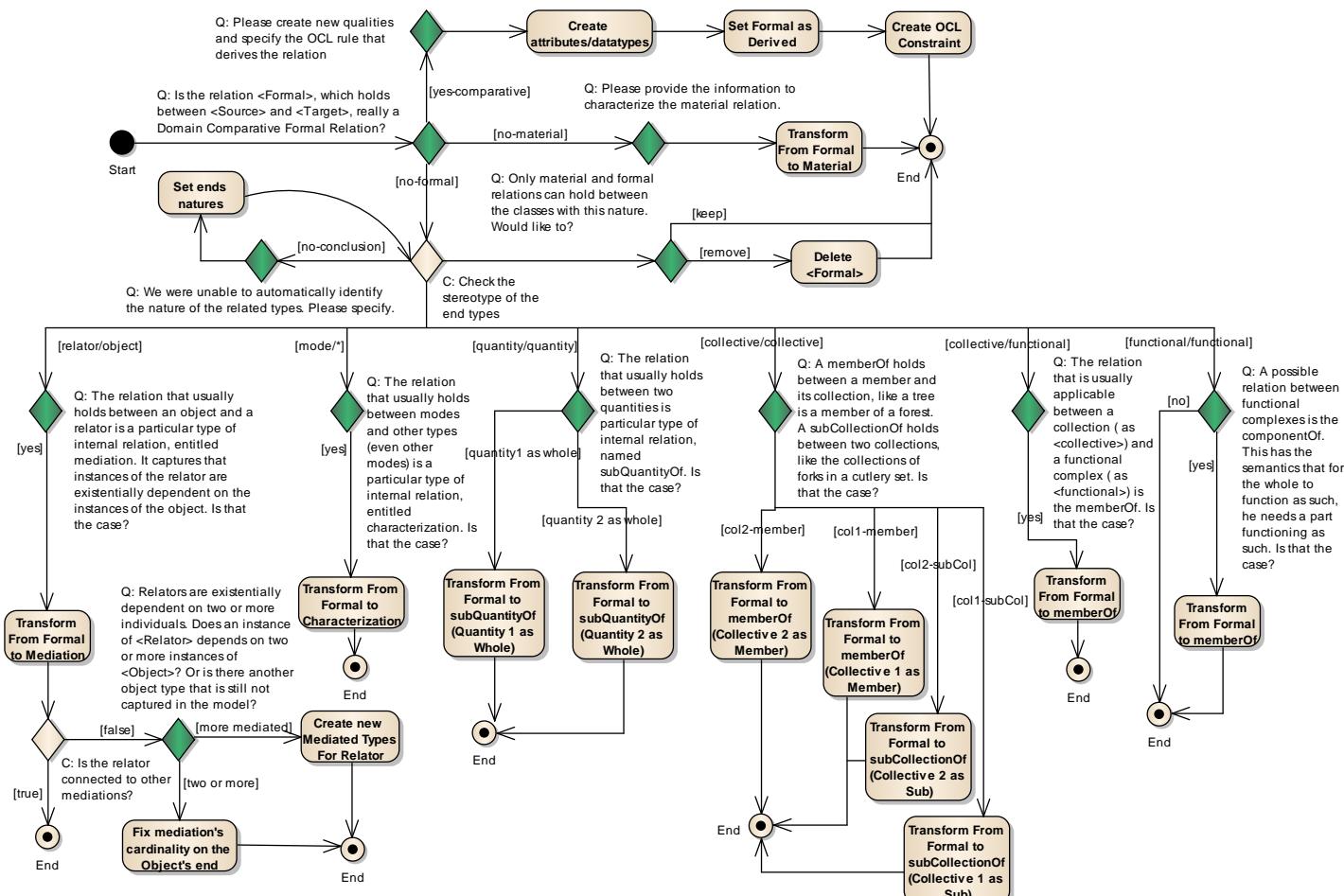
B.16 RELSPEC



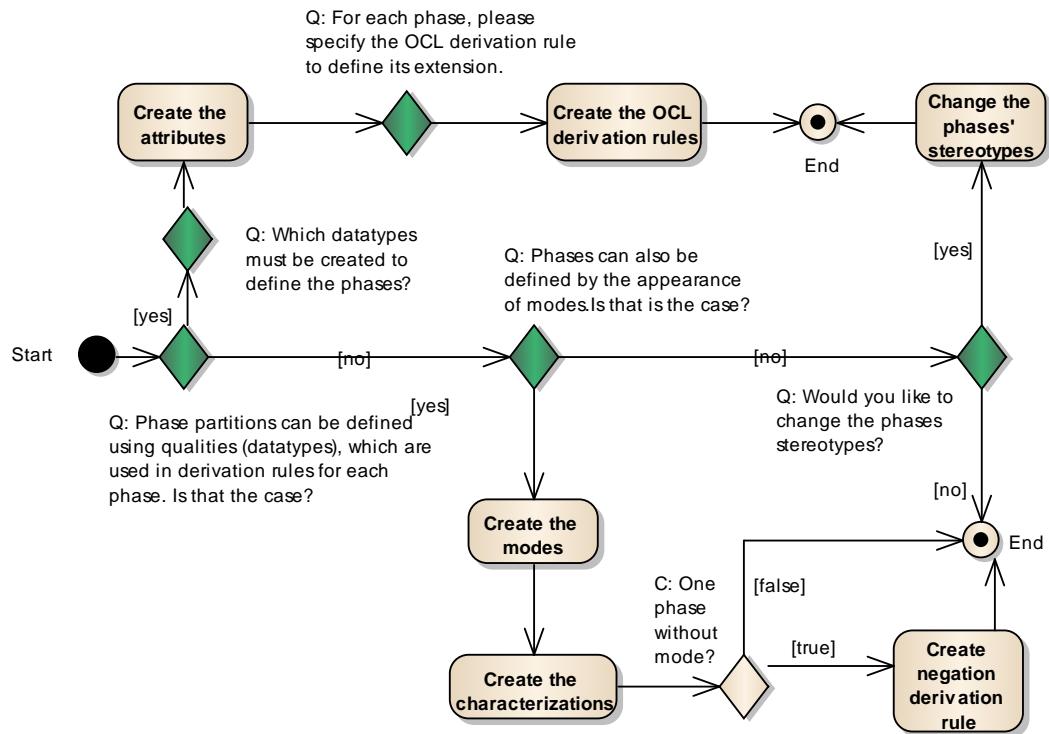
B.17 REPREL



B.18 UNDEFFORMAL



B.19 UNDEFPHASE



APPENDIX C DETAILS OF THE MGIC STUDY

The table describes the summary of participants' analysis for anti-pattern occurrences in the MGIC study. The description of the columns follows:

- “Refactoring Action” briefly describes the selected action;
- “Anti-Pattern” identify the anti-pattern type in which the action was taken;
- “Type” identifies with P the actions that were proposed by the anti-pattern and with C the ones the modeler customized the solution;;
- “Count” given the number of times the refactoring action was taken;
- “Predicted %” provides the percentage a predicted plan was selected w.r.t only the predicted solutions (for that reason it has no value for custom solutions); and
- “Error %” is the division between count and the sum of all erroneous occurrences of an anti-pattern.

Table 62. Summary of the refactoring choices for all anti-patterns.

Refactoring Action	Anti-Pattern	Type	Count	Predicted %	Error %
OCL Derivation	AssCyc	P	10	100.0%	71.4%
Invariant - Open Cycle	AssCyc	P	0	0.0%	0.0%
Invariant - Closed Cycle	AssCyc	P	0	0.0%	0.0%
Derived by intersection	DeclInt	P	14	100.0%	82.4%
Temporal constraint	DeclInt	C	2	-	11.8%
RoleMixin pattern	DeclInt	C	1	-	5.9%
Change stereotype	BinOver	P	0	0.0%	0.0%
Enforce binary property	BinOver	P	16	69.6%	51.6%
Enforce disjointness	BinOver	P	7	30.4%	22.6%
Derive role	FreeRole	P	9	42.9%	39.1%
Specialize relator	FreeRole	P	9	42.9%	39.1%
Create independent relator	FreeRole	P	8	38.1%	34.8%
Fix subtype rigidity	GSRig	P	9	60.0%	56.3%
Remove generalization from GS	GSRig	P	5	33.3%	31.3%
Delete GS	GSRig	P	1	6.7%	6.3%
Change all subtypes to mode	GSRig	C	1	-	6.3%
Change to componentOf	HetColl	P	11	100.0%	100.0%
Create functional part	HomoFunc	P	12	50.0%	36.4%
Change to memberOf	HomoFunc	P	12	50.0%	36.4%
Create inherited functional part	HomoFunc	C	8	-	24.2%
Change to non-meronymic stereotype	HomoFunc	C	1	-	3.0%
Specify subrelation to subtype	ImpAbs	P	2	66.7%	18.2%
Rule-enforced subtype restriction	ImpAbs	P	1	33.3%	9.1%
Set association as derived	ImpAbs	C	1	-	9.1%
Fix inheritance	ImpAbs	C	1	-	9.1%
Delete association	ImpAbs	C	5	-	45.5%
Transform Mixin to Sortal	Mixlden	P	10	100.0%	76.9%
Create/select new subtype	Mixlden	P	0	0.0%	0.0%
Enforce same rigidity on Mixin	MixRig	P	3	75.0%	50.0%
Change rigidity of one or more subtypes	MixRig	P	1	25.0%	16.7%
Create relator dependency	MultDep	P	2	9.1%	8.7%
Move dependency in order to new subtype	MultDep	P	2	9.1%	8.7%
Move dependency to new subtype	MultDep	P	18	81.8%	78.3%
Move dependency to ancestor's new subtype	MultDep	C	2	-	8.7%
Fix dependency multiplicity	MultDep	C	1	-	4.3%
Move dependency to new sibling	MultDep	C	1	-	4.3%
Move dependency to ancestor	MultDep	C	3	-	13.0%
Merge dependencies	MultDep	C	1	-	4.3%
Fix relator dependecy stereotype	MultDep	C	2	-	8.7%
Fix relator dependency multiplicity	MultDep	C	1	-	4.3%
Enforce composition	RelComp	P	6	100.0%	35.3%
Delete association	RelComp	C	11	-	64.7%
Enforce disjointness	RelOver	P	10	18.5%	14.3%
Enforce exclusiveness	RelOver	P	45	83.3%	64.3%

Change mediated stereotype	RelOver	C	2	-	2.9%
Create generalization	RelOver	C	6	-	8.6%
Delete mediated type	RelOver	C	1	-	1.4%
Delete mediation	RelOver	C	2	-	2.9%
Move generalization	RelOver	C	6	-	8.6%
Change mediated stereotype	RelOver	C	2	-	2.9%
Bidirectional existential dependency	RelRig	P	80	76.2%	74.8%
Create role for rigid mediated	RelRig	P	22	21.0%	20.6%
Change rigid to role	RelRig	P	8	7.6%	7.5%
Change rigid to mode	RelRig	P	2	1.9%	1.9%
Create relator subtype	RelRig	C	1	-	0.9%
Propagate change to subtypes	RelRig	C	1	-	0.9%
Subset association	RelSpec	P	196	72.3%	70.3%
Redefine association	RelSpec	P	72	26.6%	25.8%
Set association as disjoint	RelSpec	P	1	0.4%	0.4%
Delete association	RelSpec	P	5	1.8%	1.8%
Delete generalization	RelSpec	P	3	1.1%	1.1%
Change association to attribute	RelSpec	P	1	0.4%	0.4%
Reverse association	RelSpec	P	1	0.4%	0.4%
Specialize end	RelSpec	P	1	0.4%	0.4%
Enforce exclusive combination	RepRel	P	37	77.1%	64.9%
Fix mediation multiplicity	RepRel	P	15	31.3%	26.3%
Fix multiplicity on mediated end	RepRel	C	1	-	1.8%
Subset association	RepRel	C	2	-	3.5%
Modify generalization	RepRel	C	2	-	3.5%
Change mediation end type	RepRel	C	1	-	1.8%
Create relator subtype	RepRel	C	1	-	1.8%
Other	RepRel	C	2	-	3.5%
Delete formal association	UndefFormal	P	2	5.0%	4.7%
Change formal stereotype to componentOf	UndefFormal	P	26	65.0%	60.5%
Change formal stereotype to mediation	UndefFormal	P	1	2.5%	2.3%
Change formal stereotype to material	UndefFormal	P	11	27.5%	25.6%
Create relator	UndefFormal	P	11	27.5%	25.6%
Create mediation	UndefFormal	P	10	25.0%	23.3%
Set association end type as hou	UndefFormal	C	1	-	2.3%
Change to generalization	UndefFormal	C	1	-	2.3%
Set association end type as datatype	UndefFormal	C	1	-	2.3%
Change formal association to attribute	UndefFormal	C	1	-	2.3%
Enforce part disjointness	WholeOver	P	12	100.0%	75.0%
Delete part type	WholeOver	C	1	-	6.3%
Enforce part exclusiveness	WholeOver	C	4	-	25.0%