**An Approach to Design Pattern and Anti-Pattern
Detection in MOF-Based Modeling Languages**

By

Maged E. Elaasar

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment of
the degree requirements of
PH.D. in Electrical and Computer Engineering

Ottawa-Carleton Institute for
Electrical and Computer Engineering (OCIECE)

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
(19 September 2012)

Copyright ©
2012 – Maged E. Elaasar

Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-93678-8

Our file Notre référence

ISBN: 978-0-494-93678-8

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

A design pattern is a recurring and well-understood design fragment, while an anti-pattern is a common design pitfall. In the context of a modeling language, both forms of patterns are represented as structures of constrained and inter-related model elements playing complementary roles. In a model-driven engineering methodology, detecting occurrences of design patterns supports model comprehension and maintenance, while detecting anti-patterns supports model verification. With the recent explosion of domain-specific modeling languages, each with its own syntax and semantics, there has been a corresponding explosion in approaches to detecting model design patterns and anti-patterns that are tailored to those many languages. This makes developing a generic analysis tool extremely hard. Such a tool is desirable to reduce the learning curve for pattern designers as they specify patterns for different languages used to model different aspects of a system. Such a tool would also help when patterns span multiple languages, which may likely happen since several languages are often used together to model a system (e.g., UML and BPMN).

In this thesis, we propose a new unified approach to detecting both kinds of patterns for MOF-based modeling languages. MOF is an increasingly important standard that is used to define many popular modeling languages today, including UML, BPMN and SysML. In this approach, a pattern is modeled with a Visual Pattern Modeling Language

(VPML) and its occurrences in a model are then detected with a corresponding QVT-Relations (QVTR) transformation. Such a transformation runs over an input model where pattern occurrences are to be detected and reports on those occurrences in a result model. The approach is prototyped on the Eclipse platform and validated in three large case studies, two of which involve detecting design patterns—specifically a subset of the GoF patterns in a UML model and a subset of the Control Flow patterns in a BPMN model—and the third involves detecting a newly defined set of MOF anti-patterns in the UML metamodel. Results show that the approach is adequate for modeling complex design patterns and anti-patterns for MOF-based languages and detecting their occurrences with high accuracy and performance.

(I dedicate this Ph.D. thesis to my beautiful wife Nariman Nasef)

Acknowledgements

First and foremost, I would like to thank Allah Almighty for His immense blessings and help provided to me throughout my life and in particular throughout my Ph.D. journey.

I would also like to express with pleasure my gratitude to all who have contributed to this thesis. In particular, I am heartily thankful to my two supervisors, Prof. Lionel Briand and Prof. Yvan Labiche, for their incredible encouragement, help and support during the whole Ph.D. period. Their professional approach and attention to detail have helped me not only write this thesis but also produce great publications in highly rated venues. One simply cannot wish for better, more professional or friendlier supervisors.

I would also like to thank members of my family, my wife (Nariman Nasef) and three children (Mirna, Kareem and Jana), for their emotional support and encouragement, especially in tough periods of my Ph.D. journey. I am also grateful to my parents for their continuous prayers and support even though we live very far apart. I would also like to thank my brother, my sister, and other family members and friends for believing in me and for holding me to a high standard.

Finally, I would like to thank my employer, IBM Canada, for giving me the opportunity to pursue this degree part-time and for financially supporting my tuition fees. I am also thankful to my many esteemed colleagues, at IBM, at OMG, and elsewhere, for the many valuable technical discussions, contributions and reviews of my work.

Table of Contents

Abstract	iii
Acknowledgements	vi
Table of Contents	vii
List of Tables	xiii
List of Figures	xiv
Chapter 1 : Introduction	1
1.1 Problem Background: The Interest in Model-Based Technologies	1
1.2 Problem Description: Pattern Detection in Models	2
1.3 Thesis Contribution: A New Approach, Case Studies and Tools	5
1.4 Thesis Organization	8
Chapter 2 : Related Work	9
2.1 Mathematical Approaches	9
2.2 Graph Approaches	10
2.3 Quantitative Approaches.....	11
2.4 Modeling Approaches.....	12
2.5 Metamodeling Approaches	13
2.6 DSML Approaches	14
2.7 Summary	15
Chapter 3 : QVT-Relations	17
	vii

3.1 Transformation.....	17
3.2 Relation.....	18
3.3 Query.....	21
3.4 Property.....	22
3.5 Key.....	23
Chapter 4 : Pattern Specification with VPML	24
4.1 Running Example.....	24
4.2 Specifying a Pattern Catalog.....	26
4.3 Specifying a Pattern	27
4.4 Improving Specification Expressiveness	30
4.5 Improving Specification Abstraction.....	32
4.6 Dealing with Pattern Variability	33
4.6.1 Base pattern.....	33
4.6.2 Corollary Pattern.....	36
4.6.3 Parameterized Pattern	38
4.6.4 Summary of Variability Handling	39
4.7 VPML Editor	39
Chapter 5 : Pattern Detection with QVTR	40
5.1 Selection of a Transformation Language.....	41
5.2 PResults DSML	42
5.3 Mapping VPML to QVTR.....	43
5.3.1 Catalog Mapping.....	43
5.3.2 Pattern Mapping.....	45
5.3.3 Organizing Pattern Occurrences	47
5.4 Pattern Mapping Example.....	48
5.5 Detection Tooling	52
5.5.1 PResults Editor.....	52
5.5.2 VPML to QVTR Translator	53

5.5.3 QVTR Transformation Engine	53
Chapter 6 : Detecting GoF Patterns in UML—Case Study	56
6.1 Case Study Design	56
6.2 Evaluating Adequacy	58
6.2.1 Specifying Structural Conditions.....	58
6.2.2 Specifying Behavioural Conditions	59
6.2.3 Specifying Idiom Conditions	61
6.2.4 Assessing Specification Conciseness.....	62
6.3 Evaluating Accuracy.....	65
6.3.1 Setup	65
6.3.2 Model Detail Experiments	66
6.3.3 Pattern Variants Experiments	69
6.4 Evaluating Performance.....	70
6.5 Case Study Summary	71
Chapter 7 : Detecting CF Patterns in BPMN—Case Study	73
7.1 Case Study Design	73
7.2 Evaluating Adequacy	74
7.2.1 Specification Details	75
7.2.2 Assessing Specification Conciseness.....	77
7.3 Evaluating Accuracy.....	78
7.4 Evaluating Performance.....	79
7.5 Case Study Summary	80
Chapter 8 : Detecting Anti-Patterns in MOF—Case Study	81
8.1 Catalog of Metamodeling Anti-Patterns	82
8.1.1 UML Well-Formedness Anti-Patterns.....	82
8.1.2 MOF Well-Formedness Anti-Patterns	84
8.1.3 Semantic Anti-Patterns	85

8.1.4 Convention Anti-Patterns.....	86
8.2 Case Study Design	86
8.3 Evaluating Adequacy	87
8.4 Evaluating Performance.....	89
8.5 Evaluating Effectiveness.....	90
8.6 Case Study Summary.....	92
Chapter 9 : Discussion	93
9.1 Benefits and Costs of the Approach.....	93
9.2 Relevance and Applicability of the Approach	94
9.3 Relation to Other Modeling Activities.....	95
9.4 Pattern Specification Process	96
9.5 Complexity of Target Metamodel.....	96
Chapter 10 : Limitations and Future Works	98
10.1 Pattern Specification Genericity	98
10.2 Profile-Based Patterns.....	99
10.3 Unknown Variant Detection	99
10.4 Pattern Corrections.....	100
10.5 Presentation of Results.....	101
Chapter 11 : Conclusion	102
Bibliography	105
Appendix A : Complete VPML Metamodel	109
Appendix B : Summary of Mapping from VPML to QVTR	110
Appendix C : Specification of GoF Patterns on UML	112
C.1 GoF Creational Patterns	112
C.2 GoF Structural Patterns	112

C.3 GoF Behavioural Patterns	113
C.4 GoF Idiom Patterns	113
Appendix D : GMF Design Model	114
Appendix E : GoF Pattern Occurrences in GMF	115
E.1 Gold Standard Occurrences	115
E.2 Class Diagrams Only Occurrences	116
E.3 Class and Sequence Diagrams Occurrences	116
E.4 Class Diagrams and OCL Constraints Occurrences	116
E.5 All Model Details Occurrences	117
Appendix F : Specification of CF Patterns on BPMN	118
F.1 CF Basic Control Flow Patterns	118
F.2 CF Advanced Branching and Synchronization Patterns	118
F.3 CF Structural Patterns	119
F.4 CF Idiom Patterns	119
Appendix G : Specification of MOF Anti-Patterns	120
G.1 UML Well-Formedness Anti-Patterns	120
G.2 MOF Well-Formedness Anti-Patterns	121
G.3 Semantic Anti-Patterns	123
G.4 Convention Anti-Patterns	124
G.5 Examples	125
G.5.1 UML: Class With Required Owner Property Defines Another Owner	125
G.5.2 MOF: Typed Element Is Primitive With Non-Literal Default	126
G.5.3 Semantic: Association Has Asymmetric Redefinition	127
G.5.4 Convention: Operation Has Return Parameter Not Named Result	128
Appendix H : Anti-Pattern Occurrences in UML Metamodel	130
H.1 UML 2.2 Anti-Pattern Occurrences	130

H.2 UML 2.3 Anti-Pattern Occurrences	132
H.3 UML 2.4 Beta Anti-Pattern Occurrences.....	134
H.4 UML 2.4 Anti-Pattern Occurrences	135

List of Tables

Table 6-1 Statistics of the GoF specifications in VPML	64
Table 6-2 Data of initial GoF accuracy experiments with official variants ('U' means undefined)	66
Table 6-3 Data of the subsequent GoF accuracy experiments with all known variants ...	70
Table 7-1 Statistics of the CF specifications in VPML	77
Table 7-2 Data of the CF detection experiment.....	79
Table 8-1 The concrete UML metaclasses used for metamodeling with MOF 2.4.1	83
Table 8-2 UML well-formedness anti-patterns (excerpt)	83
Table 8-3 MOF well-formedness anti-patterns (excerpt)	84
Table 8-4 Semantic anti-patterns (excerpt).....	85
Table 8-5 Convention anti-patterns (except)	86
Table 8-6 Metrics of the metamodeling catalog specified with VPML.....	89
Table 8-7 Average time in seconds for running metamodeling anti-pattern categories ..	90
Table 8-8 Total number of anti-pattern occurrences in three revisions of UML.....	91
Table B-1 A summary of the mapping from VPML to QVTR.....	110

List of Figures

Figure 3-1 The UmlToRdbms transformstion	18
Figure 3-2 QVTR template for a relation	19
Figure 3-3 Some relations of the Uml2Rdbms transformation [QVT, 2011].....	20
Figure 3-4 Example of a QVTR query	22
Figure 3-5 Example of a QVTR query	23
Figure 3-6 Example QVTR keys for metaclass	23
Figure 4-1 The CLD metamodel (left) and the CLD notation (right).....	25
Figure 4-2 The half-adder design pattern (left) and the full-adder design pattern (right)	25
Figure 4-3 An excerpt from the VPML metamodel for catalog definition.....	26
Figure 4-4 An excerpt from the VPML metamodel for pattern definition	27
Figure 4-5 A basic VPML specification (top) of the half-adder pattern (bottom).....	28
Figure 4-6 Contextual properties used to generalize the half-adder pattern specification	30
Figure 4-7 A simplified and more accurate specification of the half-adder pattern	31
Figure 4-8 An abstract VPML specification (top) of the full-adder pattern (bottom).....	31
Figure 4-9 A more abstract VPML specification (top) of the full-adder pattern (bottom)	33
Figure 4-10 VPML specification of FullAdderBase pattern (left) and FullAdder1 variant (right)	34
Figure 4-11 More concise VPML specifications of the full-adder pattern variants	35
Figure 4-12 VMPL specification of FullAdder dependent pattern (left), its usage by a variant FullAdder1 as a corollary (middle), and its usage by the 2-Bit-Adder pattern as a condition (right)	37
Figure 4-13 VMPL specification of a parameterized pattern (left) & its usage by HalfAdder (right)	38
Figure 5-1 The pattern detection architecture	40

Figure 5-2 The PResults metamodel.....	42
Figure 5-3 Two representations of pattern occurrences: a tree (left) and tuples (right) ...	43
Figure 5-4 QVTR template for catalogs	44
Figure 5-5 QVTR template for patterns.....	46
Figure 5-6 QVTR keys for PResults metaclass	48
Figure 5-7 QVTR transformation detecting the full-adder pattern.....	50
Figure 5-8 A sample CLD circuit model	52
Figure 5-9 A screen shot of the PResults editor	53
Figure 6-1 Specification of the localCall operation in VPML.....	60
Figure 6-2 VMPL specifications of four GoF idiom patterns.....	62
Figure 6-3 AbstractFactory class diagram (left) and its specification in VPML (right)...	62
Figure 6-4 VPML Specification of AdapterBase (top), ClassAdapter (left), ObjectAdapter (right)	64
Figure 7-1 Example of Sequence pattern (left) and its VPML pattern specification (right)	75
Figure 7-2 Specification of some contextual properties in the CF catalog.....	76
Figure 7-3 Example of Exclusive Choice pattern (left) and its VPML pattern specification (right)	77
Figure 7-4 Specifications of the two variants of the Parallel Split pattern	78
Figure 7-5 Specification of N-out-of-M-Join idiom (left) and its usage by Discriminator (right)	78
Figure 8-1 The VPML specification of anti-pattern AssociationHasBothEndsComposite	88
Figure 8-2 The QVTR specification of anti-pattern AssociationHasBothEndsComposite	88
Figure A-11-1 The VPML metamodel for pattern definition	109

Chapter 1: Introduction

1.1 Problem Background: The Interest in Model-Based Technologies

Model-driven architecture (MDA) is an approach to the development of software systems advocated by the Object Management Group (OMG). MDA encourages the specification of system functionality as models. A model organizes information based on a metamodel. A metamodel defines the abstract syntax (i.e., the concepts and their relationships and constraints) of a modeling language. A metamodel is itself described using a modeling language called MOF [MOF, 2006] and a constraint language (based on first-order predicate logic and set theory) called OCL [OCL, 2010]. The OMG standardizes many modeling languages. Some of them are general-purpose languages, like UML [UML, 2009]. Others are domain-specific languages (DSML), like BPMN [BPMN, 2010] (for business process modeling) and MOF (for metamodeling).

Over the past decade, growing interest in MDA has led to the development of technologies and tools that generically support the definition and use of MOF-based modeling languages [Amyot et al., 2006]. A notable example of such technologies is the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009], which implements a subset of MOF called Essential MOF (EMOF). Since its inception, EMF has been used to implement numerous standard modeling languages (e.g., UML and BPMN), as well as proprietary ones. The EMF ecosystem

also provides implementations of complementary MOF technologies like query languages (e.g., OCL [MDT, 2012]), transformation languages (e.g., QVTR [Medini, 2012], QVTO [M2M, 2012], and ATL [M2M, 2012]) and graphical frameworks (e.g., GMF [GMF, 2012]). EMF is also used by several modeling tools like Rational Software Architect (RSA) [RSA, 2012] and Papyrus [Papyrus, 2012]. Because of this important and growing interest in MOF-based languages, we only focus on such languages in this thesis. We therefore always mean MOF-based (modeling) languages when we simply write (modeling) languages, and we always mean model with a MOF-based metamodel when we simply write model, unless otherwise specified.

1.2 Problem Description: Pattern Detection in Models

Another important technology for MOF-based languages is the ability to analyse models by detecting occurrences of known patterns. In this context, a pattern is a structure of constrained and interrelated model elements playing complementary roles, while a pattern occurrence is an instance of a pattern where specific model elements play the pattern roles. Patterns can be of two kinds: design patterns and anti-patterns; and we simply write pattern when we mean both kinds of patterns. A design pattern (a notion introduced to software engineering by Gamma and colleagues [Gamma et al., 1995]) is a best practice solution to a standard design problem. On the other hand, an anti-pattern [Koenig, 1995] (also known as a bad smell [Fowler, 1999]) is a common design pitfall. Each modeling language may have its own set of design patterns and anti-patterns [Booch, 2012]. For example, the Control Flow family of design patterns [Woheh et al., 2006; White, 2004] are commonly used when modeling business processes with BPMN, while the MOF set of anti-patterns [Elaasar et al., 2010] commonly arise when defining MOF-based metamodels.

In order to assist with the comprehension, maintenance and verification of models, especially for large systems, it is extremely important to be able to rely on an adequate (i.e., flexible, accurate and scalable) pattern detection technology. As models become larger and more complex, they become more difficult to comprehend. Since design patterns are well-understood fragments of design, analyzing models in terms of design patterns raises their level of abstraction, and thus helps with their comprehension. Furthermore, models are often defined by several people (with different skill sets), under tight schedules, and with continuously changing requirements [Ambler, 2002]. Without prior knowledge of existing design pattern occurrences in models, it becomes extremely difficult to preserve them when models change [France and Ghosh, 2004]. For example, if a UML model had occurrences of the GoF Observer design pattern [Gamma et al., 1995], where observer classes listen and react to changes in subject classes, it would be much easier to preserve these occurrences if they were known, or could automatically be detected, before a change is considered.

Furthermore, similar factors (e.g., tight schedules, frequent changes and lack of experience) can cause models to be infiltrated with anti-patterns of different kinds. For example, there could be syntactic anti-patterns (violations of well-formedness constraints), semantic anti-patterns (poor designs that may negatively impact model usage and implementation), convention anti-patterns (violations to methodological, organizational or project conventions), among other kinds. The automatic detection of anti-pattern occurrences can surely support the activities of model verification.

Pattern detection needs to be automated as much as possible since manually detecting them in large models cannot realistically be done and is potentially error-prone (due to the complexity

of patterns). We believe that such automation needs three prerequisites. A first prerequisite is to have patterns, both syntactically and semantically, specified in a precise machine-readable formalism. Without formal specification, patterns would be ambiguous and there would be room for confusion about their details. A second prerequisite is to be able to automatically drive detection from those specifications. It would be unrealistic to design a detection algorithm for each pattern. A third prerequisite is to be able to report pattern occurrences in a structured way. Without an efficient way to represent a possibly large number of pattern occurrences, analyzing them would not be scalable in practice.

Other desirable requirements for a pattern detection approach from an analysis tool's perspective include: 1) being applicable to any MOF-based modeling language to allow building a generic MOF-based analysis tool, 2) having an intuitive and concise syntax to ease the expression and maintenance of pattern specifications by users, 3) requiring technical skills that already exist or are easy to acquire by modeling practitioners to reduce the learning curve, 4) having some facilities (e.g., inheritance, composition) to cope with the expected variability and complexity of pattern specifications, 5) being scalable to detect and report a possibly large number of pattern occurrences in large and complex models, as this would fit the needs of industry today), and 6) reusing existing technologies to facilitate tool support (specification and detection) and integration with existing modeling tools since users would appreciate value-added features to the tools they already use rather than having to use a new one. Although many approaches to pattern detection have been proposed in the literature, our review indicated that they do not satisfy one or more of these prerequisites or requirements (Chapter 2).

1.3 Thesis Contribution: A New Approach, Case Studies and Tools

In this thesis, we propose a new approach to pattern specification and detection that satisfies the requirements described in Section 1.2. Specifically, the approach allows for specifying patterns for any MOF-based modeling language (requirement 1). MOF is widely used to define both standard modeling languages (e.g., UML and BPMN) and proprietary ones. Pattern specification is done with a newly-defined DSML called the Visual Pattern Modeling Language (VPML). The concise visual notation of VPML makes pattern specifications easy to create and maintain (requirement 2). The metamodel of VPML is small and defines concepts and relationships that should be familiar to pattern practitioners, which should make it very easy to learn and apply (requirement 3). The language has facilities (e.g., pattern inheritance/composition and contextual properties/operations) to cope with the complexity of pattern specifications (requirement 4). We describe VPML and illustrate its capabilities using a running example. We also describe the VPML tool we prototyped on Eclipse.

Furthermore, the semantics of VPML is defined by mapping it to that of QVTR [QVT, 2011], a formal and general-purpose model transformation language. In fact, one can think of VPML as a higher level and more concise alternative to specifying patterns in QVTR directly. Specifically, a scalable QVTR transformation is automatically derived from a VPML pattern specification and used to detect occurrences of the pattern (requirement 5). The transformation runs on an input model, looking for pattern occurrences, and populates a result model with details on those occurrences. The result model conforms to a newly-defined metamodel, called pattern results (PResults), which allows pattern occurrences to be reported on using a tree-based data structure. The transformation is automatically derived from a VPML specification model

using a model-to-text [M2T, 2012] mapping. We choose QVTR as it is declarative, with built-in pattern detection semantics (hence easy to map to from VPML), and a standard (hence interoperable between tools). Other transformation languages may have also been used but they would either have needed a more complex mapping of semantics (e.g., QVTO has imperative semantics closer to programming languages) or have had smaller interoperability prospects (e.g., ATL [M2M, 2012]). We used the open-source tool Medini QVT [Medini, 2012] to execute QVTR transformations on EMF-based models (requirement 6).

We validate our approach with three case studies that involve the detection of occurrences of both design patterns and anti-patterns in large models conforming to different popular MOF-based modeling languages. The first case study involves the specification of eleven representative Gang of Four (GoF) [Gamma et al., 1995] design patterns for UML (a general purpose modeling language) and detecting their occurrences in a large UML design model of an open-source software project. The second case study involves the specification of ten Control Flow (CF) design patterns [Wohed et al., 2006; White, 2004] for BPMN (a DSML) and detecting their occurrences in a large BPMN model defined for the financial services industry. The third case study involves the identification, categorization and specification of 133 MOF anti-patterns and their detection in recent revisions of the UML metamodel. The research questions we want to answer with these case studies are the following:

- 1) Is the expressiveness of VPML sufficient to apply to various patterns and MOF-based languages?

To answer this question, we use three different, complex MOF-based languages (UML, BPMN, MOF), which are standards used in the industry, we selected representative

patterns and anti-patterns of varying complexities (with multiple roles, constraints and variants) for these languages. In doing so we believe we exercised realistic conditions.

2) What is the detection accuracy of our algorithms?

To answer this question, for each of the MOF-based languages, we selected a large model in which we knew we could find a large number of pattern occurrences of various kinds and with various levels of details.

3) What is the performance of the detection mechanism?

To answer this question we selected a very large model for each of the MOF-based languages we selected. For example, in the third case study, we analyzed the UML metamodel, which is probably one of the largest MOF models that exist to date.

Results show that our approach adequately specifies complex patterns in MOF-based modeling languages. They also show that the approach detects occurrences of such patterns in large models with high accuracy and performance.

Some of our contributions were presented at the following conferences:

- Elaasar, M., Briand, L., and Labiche, Y. (2006). A Metamodeling Approach to Pattern Specification. MoDELS'06, LNCS, vol. 4199 pp.484-498.
- Elaasar, M., Briand, L., and Labiche, Y. (2011). Domain-Specific Model Verification with QVT. ECMFA'11, LNCS, vol. 6698, pp. 282-298.

A more complete description of these contributions has been submitted to a journal:

- Elaasar, M., Briand, L., and Labiche, Y. An Approach to Detecting Design Patterns of MOF-Based Modeling Languages. (Submitted to SoSyM).

1.4 Thesis Organization

The remainder of the thesis is structured as follows. Chapter 2 reviews the literature on pattern detection approaches. The definition of VPML, our Visual Pattern Modeling Language, and how it addresses common pattern specification requirements is discussed in Chapter 3. Chapter 5 illustrates the detection of patterns through a mapping from VPML to QVTR. A case study on specifying GoF patterns and detecting their occurrences in a UML model is described in Chapter 6. Chapter 7 describes a case study on specifying CF patterns and detecting their occurrences in a BPMN model. A case study on specifying MOF anti-patterns and detecting their occurrences in OMG's UML metamodel is described in Chapter 8. Chapter 9 reflects and presents insights gained from carrying out the case studies. The limitations of our approach and possible future mitigations are discussed in Chapter 10. Finally, Chapter 11 summarizes the contributions and provides the conclusions.

The thesis also has a number of appendices: Appendix A shows the complete VPML metamodel; a summary of the mapping from VPML to QVTR is given in Appendix B; Appendix C provides the VPML specification of the chosen GoF patterns for UML; The model that has been analyzed with GoF patterns is presented in Appendix D; Appendix E lists the pattern occurrences detected in the UML model; The specification of the CF patterns for BPMN is provided in Appendix F; Appendix G shows the specification of the MOF anti-patterns; Finally, the anti-pattern occurrences detected in the UML metamodel are listed in Appendix H. It is worth noting that the bulk of the material discussed in the appendices is actually published online; the appendices in these cases simply discuss the contents of the online resources and provide the URLs.

Chapter 2: Related Work

In our search for a suitable solution to the problem of pattern detection in MOF-based models, we reviewed many approaches proposed in the literature. We assessed these approaches against the prerequisites and requirements that we discussed at the end of Section 1.2. This chapter categorizes the proposed approaches, highlights notable ones in each category, discusses why they were not considered fully satisfactory, and compares them to our approach. Other good surveys of the literature in this area exist [Pettersson et al., 2009; Fulop et al., 2008; Dong et al., 2007b]. We would like to note that since we focus on pattern detection at the model level, we confine our review to approaches that operate, or can be considered to operate, at that level of abstraction. Approaches that heavily rely on analyzing lower level details in source code or that rely on runtime analysis of code execution are not reviewed here.

2.1 Mathematical Approaches

Some approaches to pattern detection use mathematical formalisms. These approaches often require a first phase where data (source code or model) is converted from its original representation into one that is more suitable for the proposed mathematical formalism (e.g., Prolog terms [Huang et al., 2005; Correa et al., 2000], a binary decision diagram [Beyer et al., 2005], a fact base [Birkner, 2007], or a Petri net [Dijkmana et al., 2008]). Many of those formalisms are based on first-order predicate logic (e.g., SQL and Prolog), while some are based

on higher-order logic (e.g., temporal logic [Mikkonen, 1998; Taibi et al., 2003; Trcka et al., 2009] and nomadic logic [Eden, 2002]). Despite their scientific contributions, we believe that these approaches have one or more of the following drawbacks. First, converting an entire model upfront from its MOF-based representation to a different one would negatively affect scalability and performance. Second, this conversion is usually specified using a different formalism (e.g., a programming language) than the one used to specify patterns, making pattern specifications more complex to create and maintain. Third, a mathematical formalism typically does not allow leveraging utility operations in metamodels, which help hide some of the complexity of those metamodels. Such operations could have been used to simplify pattern specifications. Fourth, pattern designers in the industry might not be familiar with the proposed mathematical formalisms, making them difficult to use to specify patterns.

In comparison, our approach allows specifying patterns directly on MOF-based metamodels using a graphical DSML (VPML) that is defined specifically for this purpose, making it easier to learn. VPML also incorporates technologies (like OCL for specifying constraints) that a pattern designer in the modeling domain is most probably familiar with (or can easily learn). In addition, the detection is performed directly on the MOF-based models using QVTR without a need for upfront conversion, making our approach scalable.

2.2 Graph Approaches

Like mathematical approaches, some graph-matching approaches require a first phase of converting data (e.g., source code or model) into a graph representation. For example, one approach [Meyer, 2006] converts source code into an Abstract Syntax Graph (ASG) representation and specifies anti-patterns with graph grammar rules that are applied on the ASG

by an inference algorithm. Another approach [Feng et al., 2004] represents models using an XML document and defines an anti-pattern as a template XML structure to match. Yet another approach [Seemann and Gudenberg, 1998] initially translates source code into a graph representation then applies a series of graph transformations that gradually add higher-level relationships between nodes until pattern level relationships are added. Moreover, a different approach [Eppstein, 1995; Pettersson and Lowe, 2006] represents source code as planar graphs by filtering out details that are irrelevant to the detection. It also represents patterns as graphs and uses a graph isomorphism algorithm to find matches.

In comparison, our approach can also be considered graph-based, where an analyzed model is a MOF-based graph and a VPML pattern specification is a graph template to match. However, the approach does not require a costly upfront conversion process of models into graphs. It also does not need a separate graph reduction or abstraction process as the VPML pattern specifications would directly incorporate this logic.

2.3 Quantitative Approaches

Some approaches use quantitative analysis to specify patterns. For example, one approach [Marinescu, 2004] defines metrics that detect anti-patterns. Metric values are calculated over a model and compared against thresholds. When threshold are exceeded, anti-patterns are reported. One problem here is that the choice of metrics and thresholds are always controversial. Another approach [Munro, 2005] tries to address this problem by empirically justifying the choice of thresholds. Other approaches [Dong et al., 2007a; Tsantalis et al., 2006; Gueheneuc et al., 2004] devise mathematical formulae that assign scores to both model elements and pattern roles. For example, one of the approaches [Dong et al., 2007a] assigns a unique prime number to each kind

of UML class feature (i.e., Property or Operation) and raises it to a power that equals the number of features of that kind in a class. The score for each class is the product of these numbers. Checking a pattern constraint (e.g., classes playing a certain role must have two operations) is then reduced to an arithmetic operation over these scores (e.g., whether the score for a class is a multiple of the score for the role), which can be helpful for fuzzy (near-miss) detection. However, quantitative approaches suffer from practical drawbacks. First, they are mainly suitable for encoding quantitative pattern constraints. Other constraints (e.g., the name of an operation must begin with X) are hard to encode. Second, it is not trivial for a pattern practitioner who is not trained in mathematics to devise good mathematical formulae to capture the required pattern constraints. Third, based on model size, some formulae may go outside acceptable numerical ranges.

In comparison, our approach allows the specification of any pattern constraint using either VPML (for a simple role inter-relationship) or OCL (for a more complex constraint). VPML also has facilities (e.g., pattern inheritance and composition) to cope with the variability and complexity of pattern specification. In addition, it has a scalable pattern detection algorithm encoded as a QVTR transformation.

2.4 Modeling Approaches

Other approaches use modeling techniques. They specify patterns at the (instance) model level of each modeling language. For example, one approach [Le Guennec et al., 2000; Mak et al., 2004] specifies GoF design patterns using UML collaborations. Pattern occurrences are then detected and reported as collaboration occurrences. The problem with these approaches is that what works

for one modeling language may not easily be applicable to another. Also, not all languages have, or can be extended to have, semantics for pattern specification.

In comparison, our approach specifies patterns using a separate DSML, namely VPML. VPML specifies a pattern as a template on the target language's metamodel. This makes our approach generic and applicable to any MOF-based language.

2.5 Metamodeling Approaches

Other approaches try to be generic by specifying patterns at the metamodel level. For example, one approach [Elaasar et al., 2006] augments MOF with collaboration semantics and specifies patterns with a metamodel-level composite structure diagram. Unfortunately, only a simple pattern is presented and no detection semantics are discussed, thus making it difficult to assess to which extent such an approach can work in general. Another approach [Kim and Shen, 2008] specifies patterns by extending their target metamodels with pattern-related constraints. For example, the structural aspect of GoF patterns is defined by extending the classifier-related metaclasses of the UML metamodel, while the behavioural aspect is defined by extending the interaction-related metaclasses. Pattern detection is then performed using an algorithm specific to each kind of extension; an approach that is not trivial to extend to new (DSML) extensions.

OCL has also been used as a pattern specification formalism. One approach [Enkevort, 2009] defines anti-patterns for UML class diagrams using OCL constraints and uses them to check models. One problem with using OCL to express a complete pattern is that the resulting specification becomes too complex to create and read.

In comparison, although metamodel-based, our approach allows patterns to be specified using a single high-level visual formalism (VPML) and pattern occurrences to be detected with a

single algorithm (encoded as a QVTR transformation template) that is independent of the specified patterns or the target metamodels. A VPML pattern specification only uses OCL to express constraints that are more complex than an expected attribute value for a model element.

2.6 DSML Approaches

Other approaches define new DSMLs for pattern specification. For example, one approach [Maplesden et al., 2002] defines a new DSML that focuses on structural constraints only. The approach lacks a complexity management mechanism for patterns (like composition). Another approach [Gueheneuc and Antoniol, 2008] proposes the specification of idioms (micro-patterns) with metamodels and the specification of patterns with a DSML. Detection occurs in two phases: (i) idiom models are populated by specific algorithms (written in a programming language) that query the input model, and (ii) pattern specifications are translated into a system of constraints (specific to each modeling language) that gets solved to find pattern occurrences. One issue with this approach is that specifications are encoded using several formalisms (MOF for idiom definition, source code for idiom discovery, and logic for language constraints).

Another approach [Bayley and Zhu, 2010] proposes a new metamodeling formalism based on an extension to BNF for graph definition. The formalism is used to define the abstract syntax of modeling languages. The approach also proposes a first-order logic language that works on that formalism. Finally, the approach proposes a domain-specific language for pattern specification that is specified in the same formalism. The approach has several drawbacks. First, it is not MOF-based, making it hard to integrate with MOF-based tools. Second, the metamodeling formalism has weaker semantics compared to MOF (e.g., no operations, no property subsetting or redefinition). Third, although the pattern specification language has good

variability handling mechanisms, it does not have complexity handling mechanisms (e.g., no pattern inheritance for instance). Fourth, performance has not been discussed.

Furthermore, another approach [Moha et al., 2010] proposes a rule-based language to specify anti-patterns with a way to generate detection algorithms (from rules) that demonstrated good precision and recall. Another approach [Khomh et al., 2009] is similar but converts specifications into Bayesian Belief Networks, which allow specifying probabilities on different rules, improving results as they get sorted based on confidence. However, these two approaches only allow checking pre-defined constructs and focus only on structural aspects.

In comparison, our pattern specification formalism (VPML) is MOF-based, and thus allows patterns (and their idioms and constraints) to be specified consistently and directly on MOF-based metamodels. It also has pattern complexity and variability management mechanisms (e.g., pattern composition and inheritance). In addition, its QVTR-based detection algorithm is generic (i.e., it does not need to be customized for each modeling language) and its performance has been assessed in several case studies.

2.7 Summary

Different approaches to pattern detection have been proposed in the literature, each with its own strengths and weaknesses. We reviewed and categorized many of these approaches and found them to be not fully adequate or practical as a solution to our problem. Specifically, they differ from our approach in one or more of the following ways: (i) they use specification formalisms that do not directly leverage the MOF-based representation of models; (ii) they use a second formalism to pre-process models or to specify some aspects of patterns; (iii) they use specification formalisms that are complex (e.g., the quantitative approaches), restricted (e.g.,

some approaches handle only certain constraints or languages) and/or unfamiliar to potential users (e.g., the mathematical and graph approaches); (iv) they cannot cope efficiently with pattern variability or complexity; (v) they use an un-scalable detection algorithm or require the design of a custom algorithm for each pattern or modeling language.

Furthermore, some reviewed approaches have not been validated in significant case studies [Elaasar et al., 2006] or in terms of both their accuracy and performance. In this thesis, we report on the adequacy, accuracy and performance of our approach using three large case studies (two of which are industrial). Also, many reviewed approaches have focused only on analyzing source code, sometimes by transforming it first to a user-defined model representation. In contrast, our case studies analyzed non-trivial realistic models conforming to standard metamodels. We publish all the details of our case studies online (check the appendices for URLs) to help replication and comparison.

Chapter 3: QVT-Relations

The Query/View/Transformation (QVT) [QVT, 2011] is a standard by the OMG that defines a way to transform between MOF-based models. QVT defines three transformation languages: Core, Relations and Operational. While, the first two languages are declarative (Relations is higher level than Core), the last one is an imperative language. In this chapter, we focus on describing QVT-Relations (QVTR) as we believe its syntax and semantics render themselves well to the problem of pattern detection (Chapter 5). The abstract syntax of QVTR is defined with a metamodel that extends off both MOF and OCL [QVT, 2011]. The language has two concrete syntaxes: a textual one and a graphical one. We focus on the textual syntax as it is what we map to from our VPML language. We also describe the semantics of QVTR that are relevant for our pattern detection.

3.1 Transformation

A QVTR transformation is specified between two or more MOF-based models. Each model is specified as a parameter to the transformation with a name and a type: the type is the metamodel it conforms to. For example, Figure 3-1 (line 2) shows a transformation named *UmlToRdbms* between two models, called *uml* conforming to the UML metamodel and *rdmbs* conforming to the RDBMS metamodel, a language for database schema definition. A transformation can be executed in the direction of any of the models, in which case that model becomes an output

model and all other models become input models. For example, when the *UmlToRdbms* transformation executes in the direction of the *uml* model, it transforms an input *rdbms* model (that is queried) into a corresponding output *uml* model (that is created/updated). This makes a QVTR transformation multi-directional, although typically it is executed in one particular direction.

A transformation can import another transformation (e.g., *Utilities* in line 1), in which case the importing transformation (i.e., *UmlToRdbms*) can use some of the details of the imported transformation (i.e., *Utilities*). A transformation can also extend another transformation (e.g., *BaseUmlToRdbms* in line 3), in which case it inherits all its details as its own. Those details include relations, properties and operations, which are discussed next.

```

01 imports Utilities;
02 transformation UmlToRdbms (uml:UML, rdbms:RDBMS)
03     extends BaseUmlToRdbms {
04     ...
05 }

```

Figure 3-1 The UmlToRdbms transformation

3.2 Relation

A QVTR transformation can define a set of relations between its models. A relation is a mapping (rule) between elements from these models. Figure 3-2 shows a template for a QVTR relation. Each relation is specified with a name (line 1) and a set of one or more domains (lines 2, 3 and 7). A domain is a (parameter) variable that represents either a primitive value (e.g., Integer or Boolean) (line 2) or an element from one of the transformation's models (*model1* in lines 3 and *model2* is 7). A variable is described with a name, a type, which is a metaclass from the corresponding metamodel (line 3), an optional set of attribute values, which are primitives (line 4) or references to model elements (line 5), and an optional set of conditions (line 6).

```

01 <top> relation <RelationName> {
02   primitive domain <PrimitiveName> : <PrimitiveType>;
03   checkonly domain model1 <ElementName> : <ElementType> {
04     <AttributeProperty> = <AttributeValue>,
05     <ReferenceProperty> = <ReferenceValue>:<ReferenceType> {}
06   } { <DomainCondition> }
07   enforce domain model2 <ElementName> : <ElementType> {
08     <AttributeProperty> = <AttributeValue>,
09     <ReferenceProperty> = <ReferenceValue>:<ReferenceType> {}
10   } { <DomainCondition> }
11   when {
12     <RelationPreCondition>
13   }
14   where {
15     <RelationPostCondition>;
16   }
17 }

```

Figure 3-2 QVTR template for a relation

When a domain only represents an element to match in a model, it is flagged as *checkonly* (line 3). However, when a domain also represents an element to create/update, it is flagged as *enforce* (line 7). In addition to domains, a relation can also define a number of local variables to represent calculated values (e.g. the reference value in line 5). Furthermore, a relation can have a set of pre-conditions that need to hold for the relation to hold specified in the *when* clause. It can also have a set of post-conditions that must hold once the relation holds specified in the *where* clause. Both sets of conditions can include calls to other relations in the transformation, which is a way to specify relation composition, binding values from the calling relation to the domains of the called relation. Additionally, a relation can be specified as *top* if it has to hold on its own, or non-top if it holds only when called transitively from the *where* clause of another relation.

When a QVTR transformation executes in the direction of some model *x*, all top relations execute. Such execution first involves finding all combinations of values from the input models that satisfy the input variables' expected attribute values and conditions, in addition to the relation's pre-conditions. Then for every valid combination, the output model *x* is possibly

created/updated such that values from the output variables are found to satisfy those variables' expected attribute values and conditions, along with the post-conditions of the relation.

```

01 top relation PackageToSchema {
02   checkonly domain uml p:Package {
03     name = n:String{},
04   };
05   enforce domain rdbms s:Schema {
06     name = n:String{},
07   };
08 }
09 top relation ClassToTable {
10   checkonly domain uml c:Class {
11     name= n:String{},
12     isAbstract= false,
13     package = p:Package{}
14   };
15   enforce domain rdbms t:Table {
16     name= n:String{},
17     schema = s:Schema{}
18   };
19   when {
20     PackageToSchema(p, s);
21   }
22   where {
23     AttributeToColumn(c, t);
24   }
25 }
26 relation AttributeToColumn {...}

```

Figure 3-3 Some relations of the *Uml2Rdbms* transformation [QVT, 2011]

Consider, for example, the relations of transformation *Uml2Rdbms* in Figure 3-3. Relation *PackageToSchema* (line 1) describes a mapping between two domains: a UML package *p* (line 2) and a RDBMS schema *s* (line 5). The domains require their names to be equal by assigning their name attributes to the same local variable *n* (lines 3 and 6). When this *top* relation executes in the direction of the *rdbms* model, all packages in the *uml* model are bound to variable *p* and their names are bound to variable *n*. Then, for each package, the *rdbms* model is inspected for a schema with the same name. If not found, and because variable *s* is flagged as *enforce*, the *rdbms* model is updated to have a similarly named schema.

Furthermore, relation *ClassToTable* (line 9) specifies a mapping between two domains: a UML class *c* (line 10) and a RDBMS table *t* (line 15). Both elements are constrained to have the

same name n (lines 11 and 16). Class c is additionally constrained to be non-abstract (line 12).

Both the package of class c and the schema of table t are bound to local variables p and s , respectively (lines 13 and 17). When this *top* relation executes in the direction of model *rdbms*, all non-abstract classes in the *uml* model are bound to variable c and their names are bound to variable n . Then, for each class, the *rdbms* model is inspected for a table t with the same name n . However, only those class and table combinations, whose package p and schema s are related by *Package2Schema* (line 20) are considered. In such cases, the *rdbms* model gets updated in order to have a valid binding for *enforce* variable t , followed by calling the non-top relation *AttributeToColumn* (line 23) on class c and table t .

3.3 Query

A QVTR transformation can define a set of reusable queries to help make the relations more concise. A query takes a set of input parameters and returns a result based on a body expressed in OCL. A query may also be defined with no body, in which case it is called a *black box* query. Such a query is expected to be implemented externally to the transformation in a platform-dependent way (e.g., using Java on Eclipse). Figure 3-4 (line 1) shows an example of a query called *PackageToSchema* that is extended to require package p to be top level (i.e., not have an owner). There are a few different alternatives to express this condition. The first is a direct query expression (line 5) in the *when* clause. The second is a call to a query (line 6) called *isTopLevel* that takes a package as a parameter and returns a boolean result (lines 10-12). This query can be defined once and used many times. The third is a call to a *contextual* query (line 7) that is defined in the context of metaclass *Package* and returns a boolean result (lines 13-15). (Notice

that the last form is only available for the QVT Operational language in the QVT 1.1 specification. However, we proposed adding it to QVTR as well in an upcoming revision.)

```

01 top relation PackageToSchema {
02   checkonly domain uml p:Package {...}
03   enforce domain rdbms s:Schema {...}
04   when {
05     p.owner->isEmpty(); //alternative form 1
06     isTopLevel(p); //alternative form 2
07     p.isTopLevel(); //alternative form 3
08   }
09 }
10 query isTopLevel (p:UML::Package) : Boolean {
11   p.owner->isEmpty()
12 }
13 query Package::isTopLevel () : Boolean {
14   self.owner->isEmpty()
15 }

```

Figure 3-4 Example of a QVTR query

3.4 Property

A QVTR transformation can define a set of derived properties to help make the relations more concise. A property is defined in the context of some metaclass along with a derivation expressed in OCL. The property can then be used in the same way as other properties of the metaclass. (Notice that this feature is only available for the QVT Operational language in the QVT 1.1 specification. However, we proposed adding it to QVTR as well in an upcoming revision.) A common use case for properties is to efficiently represent navigation paths (e.g., transitive closures) between model elements, leading to more concise variable definitions without having to modify the metamodel of interest. For example, consider relation *ClassToTable* in Figure 3-5 (line 1) that is extended to require the root (vs. the direct) package of class *c* to map to table *t*'s schema. One way to address this requirement is to define a new property *rootPackage* in the context of metaclass *Class* that calculates the transitive closure of all owning packages and

selects those that are top level (lines 9-11). The property can then be used to define root package p of class c (line 3).

```

01 top relation ClassToTable { ...
02   checkonly domain uml c:Class { ...
03     rootPackage = p:Package{}
04   }
05   enforce domain rdbms t:Table { ...
06     schema = s:Schema {}
07   }
08 }
09 property Class::rootPackage:Package
10 = self->closure(owningPackage)->select(isTopLevel());
11

```

Figure 3-5 Example of a QVTR query

3.5 Key

A QVTR transformation can define a set of keys for the used metaclasses. A *key* of a metaclass is a subset of that metaclass's properties that uniquely identifies its objects. By default, the notion of object identity is defined by the metamodel. Each metaclass can designate one attribute as uniquely identifying objects, and objects with the same value for that attribute are identical. However, this does not suffice for some metaclasses, since their objects are not uniquely identified by a single attribute. Figure 3-6 shows some keys defined by the Uml2Rdbms transformation. For example, different classes are allowed to have the same name only when nested in different packages, which can be represented with a key for metaclass Class consisting of both the *name* and *package* attributes (line 1). Similarly, different tables can have the same name only when defined by different schemas, which can be represented by a key for metaclass Table consisting of both the *name* and the *schema* attributes (line 2).

```

01 key UML::Class { name, package };
02 key RDBMS::Table { name, schema };

```

Figure 3-6 Example QVTR keys for metaclass

Chapter 4: Pattern Specification with VPML

In this chapter, we introduce our DSML for pattern specification, called the Visual Pattern Modeling Language (VPML). The concepts and relations defined in VPML come directly from the pattern domain hence should be straightforward to grasp by practitioners. We incrementally introduce the metamodel and notation of VPML and discuss how it can be used effectively to specify patterns, while addressing common concerns like accuracy, abstraction and variability. The complete VPML metamodel is provided for easy reference in Appendix A. In order to facilitate the presentation of our ideas, we use a running example involving the design patterns of a simple DSML. We also discuss a prototype VPML editor that we developed on Eclipse.

4.1 Running Example

The running example consists of a simple DSML we defined for circuit logic design (CLD) and two of its design patterns: half-adder and full-adder [Wikipedia, 2012]. The metamodel of CLD is shown in Figure 4-1 (left). Basically, a logic component can have a number of input pins and output pins. A gate is a component with a specific kind of logic function (AND, OR, XOR ...). A circuit is a container component for other components and wires. A wire connects exactly two pins together. The notation of the various logic elements is shown in Figure 4-1 (right).

Figure 4-1 The CLD metamodel (left) and the CLD notation (right)

Figure 4-2 The half-adder design pattern (left) and the full-adder design pattern (right)

A half-adder design pattern, shown in Figure 4-2 (left), is a circuit that adds two one-bit binary numbers. It has two input pins A and B, each representing one number, and two output pins S and C, representing the sum and the carry, respectively, of adding these two numbers. The simplest variant of half-adder connects the input pins to S, through an XOR gate, and to C, through an AND gate. Similarly, a full-adder design pattern, shown in Figure 4-2 (right), is a circuit that adds two one-bit binary numbers but accounts for values carried in as well as out. It has three input pins A, B, and Cin (representing a bit carried in from a past addition) and two output pins S and Cout (representing a bit carried out to the next addition). A full-adder can be designed in a variety of ways but a common variant is defined in terms of two half-adders by connecting A and B to the input of one half adder, connecting the sum from that to an input of the second adder, connecting Cin to the other input and finally ORing the two carry outputs.

Figure 4-3 An excerpt from the VPML metamodel for catalog definition

4.2 Specifying a Pattern Catalog

VPML allows patterns to be defined within catalogs. Figure 4-3 is an excerpt from the VPML metamodel showing class *Catalog* defining a set of patterns. For example, the two design patterns in the running example can be defined within a catalog named *Adders*. A catalog also references (through the *Catalog::metamodel* property) a set of metamodels (e.g., CLD) defining types (e.g., *CLD::Pin*) referenced by patterns within the catalog. Several metamodels may be referenced by a catalog, for instance, to define patterns that involve more than one modeling language (e.g., BPMN and UML are often used together to design a system, so one can imagine a pattern that involves both languages).

Furthermore, *Catalog* is a subclass of *EMOF::Class* and therefore a catalog can have a name. It can also define a set of attributes (of type *EMOF::Property*) and operations (of type *EMOF::Operation*) whose derivations and bodies, respectively, are expressed in OCL and used within the catalog (as will be seen later) to simplify pattern specifications. A catalog can also define a set of contextual attributes and operations (of types *ContextualProperty* and *ContextualOperation*, respectively) that are not originally part of, but are dynamically woven in, a context metaclass, without modifying its possibly standardized metamodel. One example is

catalog *Adders* defining contextual property *wiresTo:Set(Pin)*, in the context of metaclass *Pin*, with the OCL derivation expression `[self->closure(wire.pin)]` that calculates the transitive closure of other pins connected to a context pin through wires. Another example is catalog *Adders* defining contextual operation *isWiredTo(p:Pin):Boolean*, in the context of metaclass *Pin*, with the OCL body expression `[self.wiresTo->includes(p)]` that checks whether a given pin is wired to the context pin.

Finally, *Catalog* being a subclass of *EMOF::class* allows a catalog to reference other catalogs as super classes, in which case it inherits their patterns, attributes and operations. It may also reference other catalogs as imports, in which case it can access (in order to use) their patterns, attributes and operations.

Figure 4-4 An excerpt from the VPML metamodel for pattern definition

4.3 Specifying a Pattern

VPML allows patterns to be specified declaratively as a set of interrelated and constrained roles. Figure 4-4 is an excerpt from the VPML metamodel related to pattern definition. Class *Pattern* is a subclass of *EMOF::NamedElement* and therefore a pattern can have a name. It can also define

a set of one or more roles. Class *Role* is a subclass of *EMOF::TypedElement* and therefore a role can have a name and a type (a metaclass playing this role in the pattern). It can also have a set of constraints expressed with a *RoleAttribute* (specifying a value for an attribute), a *RoleInterrelationships* (specifying a value for an association end), or an OCL *condition* directly. A pattern may also have a set of OCL *conditions* that typically restricts multiple related roles.

Furthermore, a role can be flagged as exposed, which makes it visible to other patterns using the role's defining pattern (more on pattern use in Section 4.5). Conversely, a non-exposed role is not visible to using patterns. Finally, a pattern may specify that a subset of its roles is reported on. The subset is represented by property *Pattern::reportedRole* in the metamodel. It is ordered, allowing role bindings which are reported for an occurrence of the pattern to be ordered based on the relative importance of these roles (more on pattern occurrence reporting in Chapter 5).

Figure 4-5 A basic VPML specification (top) of the half-adder pattern (bottom)

We now know enough of the metamodel to start specifying the half-adder pattern (Figure 4-5 bottom) while introducing the VPML notation. The pattern *HalfAdder* (Figure 4-5 top) is depicted with a solid-line box having three compartments: a name compartment (top), a details compartment (middle) and a conditions compartment (bottom). The details compartment shows the roles involved in the pattern and how they are interrelated. Roles are represented by smaller boxes, while interrelationships are represented by arrows. Each role has a name compartment showing its name and type (e.g., A: Pin). Exposed roles are depicted with solid-line boxes, while non-exposed roles are depicted with dashed-line boxes. In this particular pattern, the circuit's pins (A, B, S and C) are exposed roles, while the intermediate details of how they are wired (i.e., wires w1-w6, gates g1-g2, and pins p1-p6) are non-exposed roles. Furthermore, a pattern shows its reported roles annotated with small number boxes in their top-left corners, representing their reporting order. For example, pin A is the first reported role and role B is the second. Some roles (e.g., g1) have a second compartment for their required attribute value (e.g., *kind = GateKind::and*). While not shown here, some roles may also have a third compartment for their OCL conditions, if any. The arrow label on a role interrelationship specifies an attribute in the source role's type that relates the two roles together (e.g., role A is related to role w1 through attribute *Pin::wire*). Traversing the arrows in this pattern from input pins (A and B) to output pins (S and C) shows exactly how they are interrelated according to the half adder pattern (Figure 4-2 left). The third compartment of a pattern shows OCL conditions that typically involve multiple roles. For example, the last condition is guarding against pin S being the same as pins p6, which is a possibility according to the metamodel (both pins are in the same collection *Wire::pin* of wire w6).

4.4 Improving Specification Expressiveness

We define the expressiveness of a pattern specification as a measure of its ability to specify different conditions or alternative specifications of a pattern, in an easy, clear, and concise manner. For example, an issue with the current specification of half-adder is that it specifies only one way of wiring the circuit's pins to the intermediate gates, which is direct wiring. In circuit design, it is possible that wires go through intermediate pins before/after being wired to the gates without affecting the function of the circuit. For example, w1 may be connected to another free pin p7, which in turn is wired to p1. The current specification will fail to detect this occurrence, an obvious limitation.

```

01 property Pin::wiresTo : Set(Pin) = self->closure(wire.pin);
02 property Pin::inputTo : Set(Component) = self.wiresTo.inComponent;
03 property Component::outputTo : Set(Pin) = self.outPin.wiresTo;
04 property Component::inputTo : Set(Component) = self.outputTo.inComponent;

```

Figure 4-6 Contextual properties used to generalize the half-adder pattern specification

In order to address this limitation, we need to change the half-adder's specification to account for any intermediate pins from the circuit's input pins (A and B) to the gates and similarly from the gates to the output pins (S and C). We can do that in VPML by first defining some contextual attributes (with OCL derivation expressions) in the catalog, as shown in Figure 4-6. Notice that attribute *wiresTo* (line 1) uses the transitive *closure* operation in OCL to find all other pins that are wired to a context pin directly or indirectly. The other attributes (lines 2-4) are convenient ones to allow bypassing the gate pins. Then, we can change the half-adder pattern specification, as shown in Figure 4-7, to connect the input and output pins directly to the gates with these new properties. For example, compare input pin A that is connected to gate g1 directly with property *inputTo*, with its counter part in Figure 4-5 that is connected to gate g1 through intermediate wire w1 and pin p1. A similar comparison can be made for output pin S, whose

connection uses property *outputTo*. The resulting half-adder specification expresses the connection conditions more accurately (covering possible intermediate wire-pin steps) and at the same time it is more concise (it has fewer roles and role interrelationships).

Figure 4-7 A simplified and more accurate specification of the half-adder pattern

Figure 4-8 An abstract VPML specification (top) of the full-adder pattern (bottom)

4.5 Improving Specification Abstraction

When patterns become complex, their specifications tend to be large and cluttered, making them hard to read and maintain. In Section 4.4, we showed how using contextual attributes helped make the half-adder specification more accurate and concise. However, consider the *FullAdder* specification (Figure 4-8 top), which corresponds to the pattern's design (Figure 4-8 bottom). Notice how the input pins in the specification are connected with contextual properties through gates g1-g5 to the output pins, in a way that parallels the circuit layout. Nevertheless, this pattern specification is still somewhat cluttered. Fortunately, VPML provides another way to abstract (i.e., make higher level by hiding details) a large pattern by using smaller patterns as conditions (pattern composition). The VPML metamodel (Figure 4-4) defines class *PatternUse*, which references a used pattern and owns a set of *RoleBindings*. Each role binding binds an exposed role (*boundRole*) of the used pattern to some role (*bindingRole*) of the using pattern. A pattern can have a set of *PatternUses* referenced as *conditionPatterns*. A condition pattern is like a pre-condition for a using pattern (i.e., the used pattern needs to hold for the using pattern to hold).

Fortunately, in the case of a full-adder, the circuit can be designed using two half-adder circuits (Figure 4-9 bottom). This allows us to refactor the full-adder pattern to use the half-adder pattern twice as a condition (Figure 4-9 top). A condition pattern-use is depicted with an ellipse showing the name of the used pattern. Role bindings are depicted with non-arrow lines connecting the pattern-use to the binding roles, while showing the name of the bound roles as labels. For example, the A and B pins of the full-adder are bound to the A and B pins of one of the half-adders. The S pin of that half-adder is bound (through intermediate pin p2) to the A pin of the other half-adder, while its C pin is bound (through intermediate pin p1) to gate g5. Notice

that the refactored full-adder specification is more abstract (higher level and less detailed), hence easier to read and maintain, than the original specification.

Figure 4-9 A more abstract VPML specification (top) of the full-adder pattern (bottom)

4.6 Dealing with Pattern Variability

4.6.1 Base pattern

Patterns may have a number of variants. In Section 4.4, we showed how to deal with one kind of variability (intermediate relations) through the use of contextual properties that specify transitive closures. Variability can also impact other kinds of pattern details (like roles, interrelationships and/or conditions), resulting in sufficiently different variants. Instead of specifying each variant separately with all its details, it makes more sense to specify common details in a separate pattern that gets used as a condition pattern by each variant. A variant then adds its own unique details on top. Since a condition pattern must hold for its using pattern to also hold (Section 4.5), the common details must hold for the specifics to hold.

For example, consider another variant of the full-adder circuit (Figure 4-2 right) where the AND and OR gates are replaced by NAND gates, without changing the overall function of the circuit. Instead of specifying this new variant from scratch, it would be desirable to specify it in a way that common details are shared with the original variant. With VPML, one can specify those common details in another pattern, named *FullAdderBase* (Figure 4-10 left), that gets used as a condition pattern by both variants. Notice that *FullAdderBase* is quite similar to the original variant (Figure 4-8) except for two differences: (i) gates g3-g5 are exposed (solid line on rectangle) and do not have their *kind* attribute value constrained (allowing such constraints to be added by the extending variants), and (ii) the exposed roles (A, B, S, C_{in} and C_{out}) are missing their little number boxes, which means they are not reported on (since *FullAdderBase* represents only a subset of the details of a pattern variant, it should not itself report any pattern occurrence). Having specified *FullAdderBase*, we now rename the original variant to *FullAdder1* and refactor it to use *FullAdderBase* as a condition (Figure 4-10 right). On top of that, *FullAdder1* adds its own specifics: it un-exposes gates g3-g5 (dashed line on rectangle), adds its specific *kind* attribute constraints (to gates g3-g5) and specifies reported roles using the little number boxes (since *FullAdder1* represents all the details of a pattern variant, it can report on occurrences of that variant).

Figure 4-10 VPML specification of FullAdderBase pattern (left) and FullAdder1 variant (right)

However, with a large number of roles in a pattern, like in *FullAdderBase*, using it as a condition is cumbersome syntactically and notationally, since an equivalent large number of role binding objects need to be created and depicted in the using patterns (Section 4.5). Fortunately, the VPML metamodel (Figure 4-4) provides a short-cut solution to this problem by allowing a pattern to directly reference (i.e., without using a condition pattern with its role bindings) another pattern as a base. Referencing a pattern as a base (syntactically through the *Pattern::basePattern* property and notationally by following the name of the pattern by “->” then the name of the base pattern) is semantically equivalent to, although syntactically and notationally more concise than, using it as a condition pattern with role bindings to similarly-named roles in the used pattern. For example, notice how the *FullAdder1* pattern in Figure 4-11 left, which references the *FullAdderBase* pattern as a base (with the “-> FullAdderBase” notation), is specified more concisely than its version in Figure 4-10 right, which uses *FullAdderBase* as a condition pattern use and binds the roles in the pattern to similarly-named roles in the condition pattern (e.g., role A in *FullAdder1* is bound to role A in *FullAdderBase*) with the ellipse/connector notation. The new variant *FullAdder2* is similarly specified (Figure 4-11 right), but requires all its gate roles to be of *kind* NAND instead of AND/OR in *FullAdder1* variant (Figure 4-11 left).

Figure 4-11 More concise VPML specifications of the full-adder pattern variants

4.6.2 Corollary Pattern

When multiple variants (e.g., P1, P2 and P3) have been specified for a pattern P, any one of them may explicitly be used as a condition of another pattern Q. This is handled by the mechanisms we have discussed so far. However, we may need to make the specification more generic and state that pattern Q requires pattern P as a condition without explicitly specifying which variant of P to use (i.e., any of P's variant can be used). In this case, the VPML variability feature described thus far (i.e., condition patterns) does not help: using all variants as conditions gives the unintended semantics of requiring them all to hold at the same time. In this section, we introduce another feature in the VPML metamodel (Figure 4-4), which can help in this situation. We first introduce the feature, and then we explain how it can help in this situation.

The feature allows a pattern to use another pattern as a corollary. While a condition pattern is like a pre-condition for using a pattern (it needs to hold for the using pattern to hold, i.e., the condition implies the using pattern), a corollary pattern is like a post condition (it holds once a using pattern holds, i.e., the using pattern implies the corollary)¹. When a pattern is used both as a condition and a corollary in the same catalog, it establishes a transitive dependency between its two using patterns (i.e., the one using it as a corollary and the one using it as a condition). For example, if pattern P is a corollary of R (i.e., *R implies P*) and also a condition of Q (i.e., *P implies Q*), then it follows from logic that R is transitively a condition of Q (i.e., *R implies Q*).

Interestingly, the corollary feature can help express the situation of generically using a multi-variant pattern as a condition without explicitly using individual variants. The idea is to define a

¹ A condition pattern-use can be compared to a relation call in the *when* clause of a QVTR relation, while a corollary pattern-use can be compared to relation call in the *where* clause of a QVTR relation (Section 3.2)

pattern *P* as a corollary of each of the variant patterns (e.g., *P1*, *P2*, *P3*). This means that *P* holds once any of the variants holds (i.e., *P1 implies P*, *P2 implies P*, ...). When pattern *P* is also used as a condition of pattern *Q* (i.e., *P implies Q*), it means that *Q* transitively holds when any variant holds (i.e., *P1 implies Q*, *P2 implies Q*, ...), which is the desired semantics.

Furthermore, a corollary is syntactically represented as a pattern-use referenced by a pattern through its *corollaryPattern* property, while notationally, it is depicted as a double-line ellipse. The used pattern in the case of a corollary must be flagged as *dependent* (i.e., it cannot hold on its own) and depicted as a double-line box.

For example, consider the CLD pattern *2BitAdder* that calculates the sum of two 2-bit binary numbers. The *2BitAdder* pattern (*Q*) is specified in VPML (Figure 4-12 right) to use the *FullAdder* pattern (*P*) twice as a condition (i.e., *FullAdder implies 2BitAdder*). Since the *FullAdder* pattern has two variants (*FullAdder1* and *FullAdder2*, Section 4.6.1), it needs to be specified as a *dependent* pattern (Figure 4-12 left) and used as a corollary of each of the variants (Figure 4-12 middle). This makes pattern *FullAdder* hold once any of the two variants holds (e.g., *FullAdder1 implies FullAdder*), which then causes *2BitAdder* to transitively hold (e.g., *FullAdder1 implies 2BitAdder*).

Figure 4-12 VMPL specification of *FullAdder* dependent pattern (left), its usage by a variant *FullAdder1* as a corollary (middle), and its usage by the *2-Bit-Adder* pattern as a condition (right)

4.6.3 Parameterized Pattern

Sometimes a pattern has little variability or one that only affects unexposed roles (hence cannot be handled using a base pattern, which requires specialized roles to be exposed). The VPML metamodel (Figure 4-4) allows a pattern to capture this variability using pattern parameters. Class *Parameter* is a subclass of *EMOF::TypedElement*, and can therefore have a name and a type (only primitive types like Integer, Boolean and an enumeration are allowed). A parameter can be used in a pattern specification by referencing it in OCL expressions. When a parameterized pattern is used by another pattern, as a condition or a corollary, the pattern-use must bind values to those parameters. Those values can themselves be expressed in OCL.

For example, consider the half-adder specification in Figure 4-7. Two input pins are wired to two gates, each of which is wired in turn to an output pin. While this design involving two gates works for the half-adder, it might also work (albeit with different gate kinds) for other circuits. In order to make such design more reusable, it can be specified as a parameterized *TwoGates* pattern, shown in Figure 4-13 (left). The pattern defines two parameters *k1* and *k2* of type *GateKind* in their own compartment below the pattern's name. The half-adder specification can then be refactored to use the *TwoGates* pattern as a condition, as shown in Figure 4-13 (right). The pattern-use specifies values for the two parameters in a compartment below the name.

Figure 4-13 VMPL specification of a parameterized pattern (left) & its usage by HalfAdder (right)

4.6.4 Summary of Variability Handling

When a pattern has small variability, such variability can be specified with parameters defined on the pattern. When the variability is large, a pattern can be specified with multiple individual variants, which reference a base pattern that has their common parts, in addition to their unique parts. When a pattern that has multiple variants needs to be used generically as a condition of another pattern (i.e., without having to use the individual variants as conditions), the pattern can additionally be specified as a corollary of each of the variants.

4.7 VPML Editor

We developed a prototype for a VPML graphical editor on Eclipse. We first defined the VPML metamodel with EMF [Steinberg et al., 2009] and automatically generated a Java-based implementation for it. Then, we used the GMF [GMF, 2012] framework to develop an MVC style graphical editor. The editor allows us to create and edit VPML models using its graphical notation. The editor was used to define the patterns of the running example and those of the case studies. While the case study patterns were specified on metamodels with open-source EMF implementations (e.g., UML and BPMN), the running example patterns were specified on CLD, for which we provided an EMF-based implementation.

Chapter 5: Pattern Detection with QVTR

In Chapter 3, we showed patterns can be specified formally in VPML. In this chapter, we show VPML specifications can be used to automate pattern detection. Figure 5-1 depicts our pattern detection architecture. It is based on mapping a VPML model to a corresponding model-to-model transformation. Such transformation would process an input model and produce an output model. The input model in this case is where pattern occurrences are detected. It conforms to a MOF-based modeling language (e.g., UML, BPMN). On the other hand, the output model is where detected pattern occurrences are reported. It conforms to a new MOF-based DSML, called pattern results (PResults), which we defined for pattern occurrence reporting. In this context, one can consider VPML as a higher level and more concise façade to specifying patterns than a model transformation language. In the remainder of this chapter, we discuss our choice of the transformation language (Section 5.1), describe the PResults DSML (Section 5.2), provide the detailed mapping from VPML to the chosen transformation language (Section 5.3), illustrate the mapping with an example (Section 5.4), and discuss the detection tool we created (Section 5.5).

Figure 5-1 The pattern detection architecture

5.1 Selection of a Transformation Language

In order to realize our architecture, we had to select a suitable model-to-model transformation language to use for executing the detection. Many such languages exist in the literature today, including QVTR, QVTO, ATL. After comparing them, we chose QVTR for a number of reasons. First, it is MOF-based, which allows expressing the transformation directly on the MOF-based abstract syntax and executing it without a need for upfront conversion from the native representation. Second, it is declarative, with its rules specified as mappings between object templates (resembling patterns) in input and output models, and hence simple to map from VPML. In contrast, QVTO has imperative semantics (like programming languages) and hence would have needed a more complex mapping. Third, QVTR is a standardized language, which increases its interoperability prospects between conforming tools. This allows creating a generic VPML tool that generates pattern detection transformations, which seamlessly integrate with different modeling tools. This is in contrast to languages like ATL, which are not standardized.

Furthermore, we note that QVTR is not widely implemented in the industry yet (although some implementations do exist like Medini QVT [Medini, 2012], which we use) due to few ambiguities in the current specification (revision 1.1), as opposed to problems with its soundness. However, we have been collaborating with the QVT revision task force by submitting issues and resolutions, based on our work, that are expected to reduce these issues in future revisions. For example, one of these issues was related to the missing support for contextual properties and operations (i.e., ones that are defined in the context of a metaclass), which is a feature in QVTO but not QVTR. In fact, this feature should have also been in QVTR and was simply omitted by mistake. We proposed adding this feature by changing QVTR's abstract syntax (metamodel) and

concrete textual syntax (EBNF grammar) to accept an optional context metaclass for attributes or operations. We also added support for this feature in Medini QVT by making its QVTR interpreter dynamically weave those contextual attributes and operations into the corresponding metaclasses. More details on QVTR can be found in Chapter 3.

5.2 PResults DSML

Pattern Results (PResults) is our DSML for representing detected pattern occurrences in a structured and scalable way. The PResults metamodel, shown in Figure 5-2, defines class *CatalogResult*, representing the detection results of a specific catalog. Each catalog result owns objects of type *PatternResult*, representing detected occurrences of a given pattern. Each pattern result owns in turn a tree of objects of type *RoleResult*, representing the bindings of pattern roles to elements playing those roles in the input model.

Figure 5-2 The PResults metamodel

Since a pattern occurrence is a unique binding of roles to elements, it follows that each branch (from root to leaf) in a *RoleResult* tree represents a unique occurrence of a pattern, as shown in Figure 5-3 (left). This can be contrasted to another representation often used in the literature [Pettersson et al., 2009], shown in Figure 5-3 (right), where pattern occurrences are

tuples of *RoleResult* objects. A direct advantage of the tree-based representation over other representations is compactness (due to ancestor branch sharing). Another advantage is that the tree makes the results easier to inspect by users as they can drill down the tree incrementally uncovering details about the detected occurrences.

Figure 5-3 Two representations of pattern occurrences: a tree (left) and tuples (right)

5.3 Mapping VPML to QVTR

In this section, we describe our pattern detection semantics through a mapping from the VPML metamodel to the textual syntax of the QVTR transformation language. We present the mapping incrementally while highlighting the relevant QVTR execution semantics. A high level summary of the mapping is given in Appendix B. For more information on the QVTR syntax and semantics we use, we refer the reader to Chapter 3.

5.3.1 Catalog Mapping

Each catalog in a VPML model maps to a QVTR transformation using the (highly abstract) QVTR template shown in Figure 5-4. The catalog's name maps to the transformation's name (line 2). The catalog's referenced metamodels (e.g., UML, BPMN) map to a (comma-separated) list of types for the transformation's *in* model parameter, while the transformation's *out* model parameter is always typed by the PResults metamodel (line 2). The list of imported catalogs

maps to the (comma-separated) list of imported transformations (line 1). Similarly, the list of catalog's super-classes maps to the transformation's (comma-separated) extends list (line 3).

Note that while a QVTR transformation is multi-directional (i.e., executable in the direction of any of its model parameters, making any of its models a possible input as well as output), the pattern detection transformation is only executed in the direction of the PResults *out* model to report the pattern occurrences detected in the *in* model.

```

01 imports <CatalogImports>;
02 transformation <Catalog> (in:{<CatalogMetamodels>}, out:PResults)
03   extends <CatalogSuperType> {
04     top relation <Pattern> {...}
05     relation <Pattern> {...}
06     query <OperationContext::><OperationSignature><{OperationBody}||>
07     property <PropertyContext::><PropertySignature><=PropertyDerivationExpression>;
08   }

```

Figure 5-4 QVTR template for catalogs

Moreover, each pattern maps to a relation in the QVTR transformation. A relation is a rule that matches elements in the input models and (possibly) updates elements in the designated output model. A relation can be of two kinds: top and non-top. When a transformation executes, all its top relations execute. On the other hand, non-top relations only execute when called transitively by other relations. Based on the VPML semantics (Section 4.6.2), a *non-dependent* pattern maps to a top relation (line 4), while a *dependent* one maps to a non-top relation (line 5).

Furthermore, recall that a catalog may define sets of attributes and operations, which are used to simplify expressions used in the catalog's patterns (Section 4.2). Each catalog's operation maps to a query in the catalog's transformation (line 6) with the same signature. The body of an operation also maps to the body of a query. When an operation does not have a body (because it cannot be specified in OCL), it maps to a body-less query (a *black box* query). If an operation is contextual to some metaclass, the query is also defined in the same context. Likewise, each

catalog's attribute maps to a property in the catalog's transformation with the same signature (line 7) and derivation expression. If an attribute is contextual to some metaclass, the property is also defined in the same context.

5.3.2 Pattern Mapping

In the previous section, we discussed that each pattern maps to a relation in a QVTR transformation. The template for such a relation is shown in Figure 5-5. In QVTR, a relation defines a correspondence between domain variables. Each domain variable represents an object to match in one of the transformation's models (in our case: *in* or *out*). The object is defined with a template consisting of a type, a set of expected property values, and a set of conditions. If a domain variable represents a primitive value (e.g., integer or string), it is flagged as *primitive*. If it represents an element to match in a model only, it is flagged as *checkonly*. If it represents an element to (also) create/update in a model, it is flagged as *enforce*. When a relation executes, it tries to bind *checkonly* domain variables to elements from their corresponding input models that conform to their templates. For each unique combination of conforming elements, the relation updates the output model of the *enforce* variables to make them bind to elements conforming to their templates as well.

Given these QVTR semantics, in our mapping, each parameter of a (parameterized) pattern maps to a *primitive* domain in the relation with the same name and type (line 2). Similarly, each exposed pattern role maps to a *checkonly* domain, in the *in* model, with the same name and type (line 3). Furthermore, each role's attribute (line 4) and role's interrelationship (line 5) value maps to a property value for a corresponding *checkonly* domain. The value in case of an interrelationship is a nested local variable denoting the interrelated role. Finally, each OCL

condition of an exposed role maps to an OCL condition on a *checkonly* domain (line 6). Non-exposed pattern roles are mapped similarly except that they are not mapped to domains (which represent the parameters of the relation) but rather to local variables defined in the relation's *when* clause (line 13) that must also bind to corresponding values from the models.

```

01 <top> relation <Pattern> {
02   primitive domain <ParameterName> : <ParameterType>;
03   checkonly domain in <ExposedRoleName> : <ExposedRoleType> {
04     <AttributeProperty> = <AttributeValue>,
05     <InterrelationshipProperty> = <InterrelatedRoleName>:<InterrelatedRoleType> {}
06   } { <RoleCondition> }
07   enforce domain out c:CatalogResult { name="<CatalogName>",
08     pattern = p:PatternResult { name="<PatternName>",
09       root = r1:RoleResult { name="<ReportedRoleName>", element=<ReportedRoleName>,
10         child = r2:RoleResult{name="<ReportedRoleName>", element=<ReportedRoleName>,
11           ...}}}};
12   when {
13     <UnexposedRoleName> : <UnexposedRoleType> {...};
14     <PatternCondition>;
15     <ConditionPattern>(<ConditionRoleBindings>;
16   }
17   where {
18     <CorollaryPattern>(<CorollaryRoleBindings>;
19   }
20 }

```

Figure 5-5 QVTR template for patterns

Moreover, the *when* clause defines extra conditions (other than variable templates) that must be satisfied on a relation before it can modify the output models. For example, the pattern's OCL conditions are mapped directly as conditions in the *when* clause (line 14). Also, the condition pattern-uses are mapped to calls to corresponding relations in the *when* clause (line 15). Each call is made with a list of arguments corresponding to the role bindings and parameter bindings of the pattern-use. The order of those arguments matches that of the domains of the called relation. Similarly, references to base patterns are mapped to corresponding relation calls, except that the call arguments bind to similarly named domains in the called relation.

Furthermore, when a pattern has roles to report, they map to an *enforce* domain (line 7). In such a case, the *enforce* domain is defined with a template that creates a pattern occurrence in the

out PResults model. The template consists of a number of nested local variables, specifically: variable *c* of type *CatalogResult* (line 7), which nests variable *p* of type *PatternResult* (line 8), which recursively nests variables r_1 - r_n of type *RoleResult* (lines 9-11) that correspond to the ordered set of *n* reported roles. Each r_i variable is assigned the name of the corresponding reported role and references that role's variable through its *element* attribute.

Additionally, when a pattern has corollaries, they are mapped to relation calls in the relation's *where* clause (line 18). This clause is executed only when conforming values are bound to the relation's domains, and once for each unique combination of these values. Only non-top relations are allowed to be called from a *where* clause. In fact, this is how they get executed since they are not executable from the transformation's top level. Each call to a non-top relation is specified with a list of arguments corresponding to the role and parameter bindings of the corollary. The order of arguments matches that of the domains of the non-top relation.

5.3.3 Organizing Pattern Occurrences

The *enforce* domain template in Figure 5-5 (lines 7-11) defines a pattern occurrence as a tree branch in a PResults model. However, for the various branches to form a tree, their elements need to share common ancestor elements, which need to be uniquely identified. This requires the transformation to look for and update objects in the output model instead of creating identical ones. By default, the notion of object identity is defined by the metamodel. Each metaclass can designate one attribute as uniquely identifying objects, and objects with the same value for that attribute are identical. However, this does not suffice for some of the metaclasses in PResults, since their objects are not uniquely identified by a single attribute. For example, different *PatternResult* objects are allowed to have the same name as long as they are nested under

different *CatalogResult* objects. Similarly, *RoleResult* objects can have the same *name* and *element* combinations only when nested in different ancestor objects.

To configure our QVTR transformation to create a tree with unique branches of PResults objects, we need to (i) tell it which combination of attribute values make each metaclass unique, and (ii) include attributes designating ancestors in these combinations. Fortunately, QVTR provides a feature called metaclass *keys*. A *key* is a subset of the metaclass's attributes that uniquely identifies its instances. In this case, we need to define *keys* for the PResults metaclasses (Figure 5-2) that also include their ancestor attributes, as shown in Figure 5-6. This configures the transformation to create new PResults objects only if similar ones do not already exist under the same ancestor. Notice that class *RoleResult* has two keys (lines 3-4), since it can be nested either by a *PatternResult* (for root role results) or a *RoleResult* (for nested role results). In this case, the transformation will look for *RoleResult* objects using either of these two keys.

```
01  key CatalogResult { name };
02  key PatternResult { catalog , name };
03  key RoleResult { pattern, name, element };
04  key RoleResult { parent, name, element };
```

Figure 5-6 QVTR keys for PResults metaclass

5.4 Pattern Mapping Example

In this section, we illustrate the mapping of a VPML model to QVTR. Specifically, Figure 5-7 shows the QVTR transformation for the full-adder pattern, specified earlier in VPML (*FullAdderBase* in Figure 4-10 and *FullAdder/FullAdder1* in Figure 4-12), using the mapping rules in Section 5.3. First, the *Adders* catalog in the VPML model (Section 4.2) maps to a QVTR transformation with the same name (line 1) that is specified between an *in* CLD model (referenced by the catalog) and an *out* PResults model.

The *FullAdderBase* pattern (Figure 4-10) is mapped to a QVTR top relation with the same name (line 2). The exposed roles of the pattern (i.e., *A*, *B*, *Cin*, *S*, *Cout*, *g3*, *g4* and *g5*) are defined as (*in* model) checkonly domain variables with the same names and types (lines 3-10). The interrelationships of those roles map to attribute value constraints for the corresponding domain variables. For example, the *inputTo* interrelationships from role *A* to roles *g1/g4* are mapped as values *g1/g4* for variable *A*'s *inputTo* attribute (line 3). On the other hand, the pattern's unexposed roles (i.e., *g1* and *g2*) are mapped to local variables defined in the relation's *when* clause (lines 12-13) that also have attribute values corresponding to their interrelationships. Additionally, these roles have attribute values corresponding to their *kind* attribute constraints. For example, variable *g1* has value *GateKind::xor* for its *kind* attribute (line 12).

Furthermore, the *FullAdder1* pattern (Figure 4-12 middle) is mapped to a top relation with the same name (line 16). The exposed roles of this pattern (i.e., *A*, *B*, *Cin*, *S* and *Cout*) are mapped to (*in* model) checkonly domain variables with no attribute values (lines 17-21). Similarly, the unexposed roles (i.e., *g3-g5*) are mapped to local variables in the *when* clause with no attribute values (lines 23-25). Since *FullAdder1* references *FullAdderBase* as a base pattern, its *when* clause has a call to relation *FullAdderBase* with a list of arguments that binds the domain variables of *FullAdderBase* to similarly named variables in *FullAdder1* (line 26). Additionally, since *FullAdder1* references *FullAdder* as a corollary pattern (Figure 4-12 middle), its *where* clause has a call to relation *FullAdder* with a list of arguments that bind the domain variables of *FullAdder* to similarly named variables in *FullAdder1* (line 29).

Finally, the *dependent* pattern *FullAdder* (Figure 4-12 left) is mapped to a non-top relation with the same name (line 34). The exposed roles of this pattern (i.e., *A*, *B*, *Cin*, *S* and *Cout*) are

mapped to (*in* model) *checkonly* domain variables with no attribute values (lines 33-37). Since the pattern reports on those exposed roles, the roles also map to an (*out* model) *enforce* domain variable *c* of type *CatalogResult* (line 38), which nests variable *p* of type *PatternResult* (line 39), which recursively nests variables *r1-r5* of type *RoleResult* (lines 40-44). Each *ri* variable is assigned the *name* of the corresponding role and references it through its *element* attribute.

```

01 transformation Addders (in:CLD, out:PResults) {
02   top relation FullAdderBase {
03     checkonly domain in A : Pin { inputTo = g1:Gate {}, inputTo = g4:Gate {} };
04     checkonly domain in B : Pin { inputTo = g1:Gate {}, inputTo = g4:Gate {} };
05     checkonly domain in Cin : Pin { inputTo = g2:Gate {}, inputTo = g3:Gate {} };
06     checkonly domain in S : Pin {};
07     checkonly domain in Cout : Pin {};
08     checkonly domain in g3 : Gate { inputTo = g5:Gate {} };
09     checkonly domain in g4 : Gate { inputTo = g5:Gate {} };
10     checkonly domain in g5 : Gate { outputTo = Cout:Pin {} };
11     when {
12       g1:Gate { kind = GateKind::xor, inputTo = g2:Gate {}, inputTo = g3:Gate {} };
13       g2:Gate { kind = GateKind::xor, outputTo = S:Pin {} };
14     }
15   }
16   top relation FullAdder1 {
17     checkonly domain in A : Pin {};
18     checkonly domain in B : Pin {};
19     checkonly domain in Cin : Pin {};
20     checkonly domain in S : Pin {};
21     checkonly domain in Cout : Pin {};
22     when {
23       g3:Gate { kind = GateKind::and };
24       g4:Gate { kind = GateKind::and };
25       g5:Gate { kind = GateKind::or };
26       FullAdderBase(A, B, Cin, S, Cout, g3, g4, g5);
27     }
28     where {
29       FullAdder(A, B, Cin, S, Cout);
30     }
31   }
32   relation FullAdder {
33     checkonly domain in A : Pin {};
34     checkonly domain in B : Pin {};
35     checkonly domain in Cin : Pin {};
36     checkonly domain in S : Pin {};
37     checkonly domain in Cout : Pin {};
38     enforce domain out c:CatalogResult { name="Addders",
39       pattern = p:PatternResult { name="FullAdder",
40         root = r1:RoleResult { name="A", element=A,
41           child = r2:RoleResult { name="B", element=B,
42             child = r3:RoleResult { name="Cin", element=Cin,
43               child = r4:RoleResult { name="S", element=S,
44                 child = r5:RoleResult { name="Cout", element=Cout
45               }}}}}}};
46   }
47 }

```

Figure 5-7 QVTR transformation detecting the full-adder pattern

We now describe what happens when the transformation in Figure 5-7 is executed on a sample input CLD model, like the one shown in Figure 5-8. Since the transformation has two top relations (i.e., *FullAdderBase* and *FullAdder1*), the execution engine decides which relation to check first. In this case, since *FullAdder1* uses *FullAdderBase* as a condition (line 26), the latter is checked first. As relation *FullAdderBase* has five input variables of type Pin (A, B, S, Cin, Cout) and five variables of type Gate (g1-g5), and since the input model has 20 pins (the black dots) and five gates, the execution engine detects all sets of valid bindings that satisfy the input variable conditions (refer to Section 5.5.3 for a description of how the execution engine can do this efficiently). Since this relation includes all the connection conditions, the engine finds only one valid set that includes the following bindings: X1 for Cin, X2 for A, X3 for B, Y1 for Cout, Y2 for S, Z4 for g3, Z3 for g4, and finally Z5 for g5.

Once the engine is done checking *FullAdderBase*, it then checks *FullAdder1*, which has the same domains as *FullAdderBase*. However, since *FullAdderBase* is a condition for *FullAdder1* (line 26) that involves binding all its domains to those of *FullAdder1*, the execution engine can efficiently narrow down the candidate bindings to the single set of bindings obtained for *FullAdderBase*. It additionally verifies the other unique conditions placed on variables g3-g5 (lines 23-25) and finds them to hold in this case as well.

After that, the engine turns to the *where* clause of *FullAdder1* and checks relation *FullAdder* (line 29), whose domains are bound to ones in *FullAdder1*. Since *FullAdder* does not add any new conditions to the domains, the same single set of bindings from *FullAdder1* applies to this relation as well. The engine then turns to enforcing the output domain of *FullAdder* (lines 38-45), which leads to creating a PResults pattern occurrence corresponding to that set of bindings.

Figure 5-8 A sample CLD circuit model

5.5 Detection Tooling

In this section, we describe the tools we developed (or used) for detecting patterns specified in VPML, which are: (i) the PResults editor for inspecting the detected pattern occurrences, (ii) the VPML to QVTR translator to automatically generate a detection transformation and (iii) the execution engine of the resulting QVTR transformations using Medini QVT. We also discuss some of the engineering challenges we faced and how we resolved them.

5.5.1 PResults Editor

We used EMF to define the PResults metamodel and generate a corresponding Java-based implementation for it. Then, we developed a tree-based editor for PResults models (illustrated on an example in Figure 5-9). We made the nodes in the tree show the number of pattern occurrences below them since we believe this would facilitate result inspection. Specifically, it can help a user, inspecting the detection results (by drilling down the tree), focus on the branches with the most occurrences for instance.

Figure 5-9 A screen shot of the PResults editor

5.5.2 VPML to QVTR Translator

We developed a VPML to QVTR translator using JET [M2T, 2012], which is an EMF-based model-to-text transformation framework on Eclipse, thereby making the generation of QVTR transformations from VPML specifications automated. (This is how Figure 5-7 was created.) JET provides a declarative way of specifying model-to-text transformations and was chosen due to our previous familiarity with it. Alternative frameworks on Eclipse, like Acceleo [Acceleo, 2012], could have also been used.

5.5.3 QVTR Transformation Engine

We used the open-source Medini QVT tool [Medini, 2012] to inspect, execute and debug the resulting QVTR transformation. Early execution experiments with Medini pointed to a scalability problem with its implementation when analyzing large models. We investigated and identified two causes: 1) repetitive evaluation of the transformation's queries and 2) a slow variable binding algorithm for a relation. We improved the Medini implementation to address

both issues (the improvements will be submitted to the open source project at some point). We addressed the first issue by caching the results of query evaluations, using their arguments for identification, since such results do not change (because the input model where we look for pattern instances does not change). Before performing a new evaluation, the cache is inspected for a previous result. We addressed the second issue by improving the variable binding algorithm. The problem was coming from the way the variable binding tree was constructed and the order in which conditions on the variables were checked along the tree branches. With a declarative language, such as QVTR, the order in which variables and conditions are specified should be irrelevant, and an execution algorithm should be able to analyze variables and conditions inter-dependencies to decide on an efficient way to setup the variable binding tree to help prune its branches as early as possible.

On the contrary, Medini's algorithm setup the variable binding tree without such an analysis and variable binding was often purely based on the syntactical order in which variables and conditions appeared in the relation. For example, all variables of relation *FullAdderBase* (Figure 5-7, lines 2-15) were bound at the root of the tree (i.e., their candidate values included all objects of their types in the model leading to a huge search space for a large model), overlooking the fact that some of them, like variable *g1* and *g4* (lines 3-4), were bound as attribute values of other variables, which meant they could have had a much more restricted set of candidate values. Another example is the condition on the *kind* attribute of variable *g3* (line 13) that was checked towards the bottom of the binding tree (as it appeared towards the end of the relation) and not directly after variable *g2* was bound (line 5), which could have helped prune the tree much earlier.

We addressed the scalability of the variable binding algorithm with three main improvements. First, we analyze the inter-dependencies between variables and construct a depth-first search tree where less dependent variables are bound earlier in the tree than more dependent variables. For example, this makes variables A , B and C_{in} of relation *FullAdderBase* pop to the root of their tree as they are not at all dependent on other variables. Having a small number of root variables to bind can dramatically improve the performance of a search tree as they reduce the number of initial candidate bindings. Second, we analyze the inter-dependencies between variables and conditions and use that information to check conditions directly after all their dependent variables are bound in the tree. For example, the *kind* value condition on variable $g3$ (line 13) would be checked immediately after the variable was bound (line 5). This causes the search space to be effectively pruned as early as possible. Third, we use the relation calls in the *when* clause (e.g., the call to relation *FullAdderBase* from the *when* clause of relation *FullAdder1*, line 26) to reduce the number of candidate bindings early as well. Since these called relations need to have been bound before their calling relations can execute, the set of successful variable bindings for them are already known, and hence can be used to reduce the candidate bindings of the corresponding variables in the calling relations.

Chapter 6: Detecting GoF Patterns in UML—Case Study

The applicability of our approach (specifying modeling patterns with VPML and detecting them with QVTR) was assessed in a case study using three practical criteria: adequacy, accuracy and performance. Adequacy assessed the ability of the approach to specify a realistic family of design patterns that features multiple variants and different kinds of conditions. Accuracy assessed the ability of the approach to detect pattern occurrences accurately (as measured by both precision and recall metrics). Since a model represents an abstraction of a system, it may be defined at different levels of details. We suspected that pattern detection might be sensitive to the level of model detail, and hence wanted to measure the impact of different levels of details on accuracy. We also knew that patterns in practice have variants. Hence, we also wanted to measure the impact of accounting for such variability on accuracy. Performance assessed the ability of the approach to detect pattern occurrences in a scalable fashion. Since it was not clear how much impact trying to optimize accuracy has on performance, we also tried to measure it.

6.1 Case Study Design

The case study involved detecting a representative subset of the Gang of Four (GoF) design patterns [Gamma et al., 1995] in a UML model. The subset consisted of 11 (out of 23) GoF patterns in three different categories: (i) Creational (Singleton, Abstract Factory and Factory Method), (ii) Structural (Adapter, Bridge, Composite and Decorator), and (iii) Behavioural

(Chain of Responsibility, Command, Observer and Strategy). We consider this was sufficient to evaluate the adequacy of our approach to specify different kinds of patterns. The analyzed UML model (Appendix D) reflected the design of a large open-source project called the Graphical Modeling Framework (GMF) [GMF, 2012] and was created by one of its architects². The model was interesting to analyze since GMF was known to have used GoF patterns extensively, and was also relatively large (four packages, 95 classes, 127 attributes, 440 operations, 109 interactions, 500 messages and 189 OCL constraints).

Furthermore, UML design models can describe the structure as well as the behaviour of a system. This is why the GoF patterns are often described in terms of both aspects. However, in practice, a UML model may not have all the necessary behavioural information to accurately detect GoF patterns. As one of the stated objectives of this case study is to assess the accuracy of detection for different levels of model detail, we thought of assessing the impact of different levels of behavioural specification in a UML model on the accuracy of detecting GoF patterns.

Since the GMF model we used defined structural aspects with UML class diagrams and behavioural aspects with UML sequence diagrams and OCL constraints, we ran four experiments where we setup different levels of behavioural information in the model: (level 1) class diagrams only, (level 2) class and sequence diagrams, (level 3) class diagrams and OCL constraints, and (level 4) class diagrams, sequence diagrams and OCL constraints. We also wanted to assess the accuracy of the approach when different pattern variants are considered. Therefore, for each of the four experiments (focusing on a specific level of model detail), we ran two sub experiments where the pattern specifications considered: (i) the official GoF variants only; (ii) all known GoF

variants (to the authors). In addition, for each of these sub-experiments, we also measured the performance of detection.

6.2 Evaluating Adequacy

In this section, we assess the adequacy of VPML to specify the chosen subset of GoF patterns on UML. (We would like to note that our specification of this subset is by no means authoritative. It is rather based on our own interpretation of the informal description of this subset in the literature.) The subset features a variety of conditions including: (i) structural conditions (i.e., conditions on the structure of the UML model as specified by class diagrams), (ii) behavioural conditions (i.e., conditions on the behaviour of the UML model as specified by sequence diagrams and OCL constraints), and (iii) idiom conditions (i.e., small patterns that are often used by larger patterns as conditions). We show how VPML can be used to specify these kinds of conditions, followed by an assessment of the overall conciseness of the VPML catalog for GoF patterns, which is shown in details in Appendix C.

6.2.1 Specifying Structural Conditions

Structural conditions specify how UML elements are inter-related in structural diagrams. This kind of conditions was found in all GoF patterns, especially those in the Structural category. A structural condition mainly took the form of an expected value for a role's attribute or interrelationship. For example, the Product role of the AbstractFactory pattern was played by an abstract class, and hence had a *RoleAttribute* element indicating that the *isAbstract* attribute must have a value of *true*. Another example was the Leaf class of the Composite pattern, which is a

² The architect is Mohamed Mostafa who led the GMF project at IBM before it was contributed to Eclipse.

subclass of the Component class, and hence its role was specified with a *RoleInterrelationship* to the Component role referencing the *superClass* association end. However, some structural conditions took different forms and hence needed to be specified in OCL. For example, condition `[Leaf.superClass->excludes(Composite)]` was specified to exclude class Composite from being a super class of Leaf in the Composite pattern.

In most cases, the properties referenced by structural conditions were defined by the UML metamodel. However, a couple of contextual properties representing transitive closures were needed to generalize some patterns. For example, we defined the contextual property *Class::allSuperClasses:Set(Class)* with derivation expression `[self->closure(superClass)]` to represent the transitive closure of all super classes of a given class. The property was used in the AbstractFactory pattern, for example, to relate class ConcreteProduct to class Product.

6.2.2 Specifying Behavioural Conditions

Behavioural conditions restrict the behaviour of UML elements as defined by behavioural diagrams and OCL post conditions. This kind of conditions was found in most GoF patterns, especially those in the Behavioural category, and took one of three forms: (i) a call to an operation on the same object, (ii) a call to an operation on a different object, and (iii) an operation creating objects of some class. These conditions were specified in the catalog with three Boolean operations (*localCall*, *delegationCall* and *objectCreation*, respectively). The operations analyzed two kinds of behavioural information in the UML model: interactions (sequence diagrams) and (OCL) post-conditions of operations.

For example, consider the VPML specification of the *localCall* operation (Figure 6-1). The operation (lines 1-3) checks whether the *caller* UML operation calls the *callee* UML operation

on the same object by delegating to another operation *transLocalCall* (lines 4-8) that checks whether *caller* calls *callee* either directly (line 5) or transitively via another operation owned by the class (lines 6-7). Operation *directLocalCall* (lines 9-13) first checks if *caller* has any post conditions or interactions (called methods) (line 10). If it has neither (which is often the case when the model is under specified), then it returns *true* anyway, which is an approximation that reduces false negatives but may increase false positives. However, if *caller* has any of them then it investigates further by checking the interactions (line 11) and post-conditions (line 12).

```

01 operation localCall(caller:Operation, callee:Operation) : Boolean {
02   transLocalCall(caller, callee, Set{caller})
03 }
04 operation transLocalCall(caller:Operation, callee:Operation, stack:Set(Operation)):Boolean{
05   directLocalCall(caller, callee) or
06   caller.class.ownedOperation->exists(o| stack->excludes(o) and
07     directLocalCall(caller, o) and transLocalCall(o, callee, stack->union(Set{o})))
08 }
09 operation directLocalCall(caller:Operation, callee:Operation) : Boolean {
10   (caller.postcondition->isEmpty() and caller.method->isEmpty()) or
11   interactionLocalCall(caller.method.oclassType(Interaction), callee) or
12   caller.postcondition->exists(c| postConditionLocalCall(c, callee))
13 }
14 operation interactionLocalCall(interaction:Interaction, callee:Operation) : Boolean {
15   interaction.message->exists(m|
16     m.messageSort=MessageSort::synchCall and
17     m.signature = callee and
18     m.sendEvent.oclassType(MessageOccurrenceSpecification).covered->exists(name='self') and
19     m.receiveEvent.oclassType(MessageOccurrenceSpecification).covered->exists(name='self'))
20 }
21 operation postConditionLocalCall(c:Constraint, callee:Operation):Boolean;

```

Figure 6-1 Specification of the localCall operation in VPML

Checking interactions is directly specified over the UML metamodel by operation *interactionLocalCall* (lines 14-20). It looks for a synchronous message in the interaction whose signature matches that of operation *callee* (line 15) and that starts from a lifeline that represents the *self* variable (line 16) and that ends at a lifeline that also represents the *self* variable (line 17).

Checking OCL post-conditions, on the other hand, cannot directly be specified in VPML. This is because the OCL expressions in the UML model are encoded using their textual (as opposed to abstract) syntax, which prevents their introspection (e.g., looking for a call to a given

operation) using OCL directly. Instead, we specified this checking using a bodyless operation *postConditionLocalCall* (line 21), which gets mapped to a QVTR *black box* query (Section 5.3.1), and is implemented in a platform dependent way (in this case, in Java using the Eclipse OCL API, which allows inspecting the textual syntax of OCL expressions).

6.2.3 Specifying Idiom Conditions

The GoF patterns were previously found to use some common idioms [Smith, 2005]. We identified four of these idioms that we considered to be building blocks for the other idioms and specified them as patterns in VPML (Figure 6-2). The first is *Redefinition*, whereby an operation in one class is redefined by a subclass. The second is *Conglomeration*, whereby an operation in one class is locally called from another operation in the same class or a subclass. The third is *Delegation*, whereby an operation in one class is called by an operation in another class using one of its properties. The fourth is *Creation*, whereby an operation creates objects of some class. We used these idioms as conditions in many GoF specifications to make them concise and avoid duplicating conditions.

For example, Figure 6-3 (right) shows the VPML specification of the *AbstractFactory* pattern, whose class diagram is shown in Figure 6-3 (left). The pattern provides an interface *Factory* that allows class *Client* to create instances of class *Product* without committing to its concrete class. Then, a *ConcreteFactory* class, conforming to *Factory*, can create instances of class *ConcreteProduct* that conforms to *Product*. The VPML specification of this pattern uses three of the four idioms introduced above as conditions. First, it uses the *Delegation* idiom to specify that an *operation* in class *Client* invokes the *createProduct* operation on the *Factory* interface. Second, it uses the *Redefinition* idiom to specify that the *createProduct* operation of

the *Factory* interface is redefined by class *ConcreteFactory*. Third, it uses the *Creation* idiom to specify that the *createProduct* operation of the *ConcreteFactory* class creates instances of the *ConcreteProduct* class.

Figure 6-2 VMPL specifications of four GoF idiom patterns

Figure 6-3 AbstractFactory class diagram (left) and its specification in VPML (right)

6.2.4 Assessing Specification Conciseness

The complete VPML catalog of the chosen subset of the GoF patterns is provided in detail in Appendix C. However, we show in Table 6-1 some statistics collected from the catalog to help the reader assess its conciseness. While we specified at least one variant for each pattern, some

had more variants (e.g., two for Singleton), in which case we also specified a base pattern. The last three columns count the number of roles, idioms and conditions for each variant, as well as for the base pattern if any (in parenthesis).

One observation is that each specification used 1-3 idioms and 1-5 conditions (including attribute values, interrelationships and OCL conditions). Given that the idioms themselves used an average of four conditions (e.g., the Redefinition idiom in Figure 6-2 used four interrelationships), the use of idioms in this catalog reduced the number of conditions per pattern, and hence enhanced overall conciseness by 68% (if we measure conciseness as the number of conditions being used in a pattern): the number of idiom occurrences in all patterns/variants (the total of the idioms column in Table 6-1, i.e., 27), multiplied by the number of conditions saved per idiom (i.e., 4), all divided by the sum of that number (i.e., $4 * 27$) and the number of other conditions used by the patterns (the total of the conditions column in Table 6-1, i.e., 52). One exception was the *Singleton* specification, which had a small number of roles and several conditions that did not fit into any of the idioms.

When patterns had variants, they were specified by capturing their common roles and conditions in base patterns, referenced by each variant, along with a small number of additional conditions. For example, consider the VPML specification of the Adapter pattern with two variants (ClassAdapter and ObjectAdapter) in Figure 6-4. The base pattern AdapterBase (top) has a total of four roles (*Target*, *Adapter*, *request1* and *request2*), uses one idiom (*Redefinition*) and specifies two conditions (on *Target* and *request*). The ClassAdapter variant (left) adds one extra role (*realRequest*) for a total of five, one extra condition (on *realRequest*) and one extra idiom (*Conglomeration*). Similarly, the ObjectAdapter variant (right) adds two extra roles

(*realRequest* and *anAdaptee*) for a total of six, one extra condition (on *anAdaptee*) and one extra idiom (*Delegation*).

Table 6-1 Statistics of the GoF specifications in VPML

Category	Pattern	Reported Roles	Variants	Roles	Idioms	Conditions
Creational	Abstract Factory	Factory, Product	1	9	3	5
	Factory Method	Creator, Product	1	7	3	2
	Singleton	Singleton	2	(2) 3,2	(0) 0,0	(6) 5,1
Structural	Adapter	Target, Adapter, Adaptee	2	(4) 5,6	(1) 1,1	(2) 1,1
	Bridge	Abstraction, Implementation	1	7	2	2
	Chain of Resp.	Handler	1	5	3	1
	Composite	Component, Composite	1	7	1	1
	Decorator	Component, Decorator	2	(10) 5,5	(2) 1,1	(2) 1,3
Behavioural	Command	Command, ConcreteCommand, Receiver	1	10	3	5
	Observer	Subject, Observer	2	(7) 7,5	(2) 1,0	(3) 4,5
	Strategy	Context, Strategy	1	7	2	2

Figure 6-4 VPML Specification of AdapterBase (top), ClassAdapter (left), ObjectAdapter (right)

6.3 Evaluating Accuracy

6.3.1 Setup

We carried out a set of experiments to assess the accuracy of detecting GoF occurrences using two standard metrics: precision (the ratio of correctly found to all found occurrences) and recall (the ratio of correctly found to all correct occurrences). We recorded the set of all correct occurrences in the model in a gold standard (GS). These occurrences were found by manually inspecting the model and by interviewing the GMF architect who designed the case study system. Since accuracy calculations depend on the numbers of unique occurrences, which in turn depend on the roles reported for each pattern, we developed a criterion for choosing the reported roles. Specifically, we chose roles that were played by classes that were heads of their inheritance hierarchies or related to them by association. This criterion selected primary roles (Table 6-1 - third column) that often sufficed, for someone familiar with the model, to judge the validity of GoF occurrences without doing deeper inspection. For example, for the Adapter pattern, using the criterion led us to select the Target and Adapter classes (since they are at the top of their inheritance hierarchy) and the Adaptee class (since it is related to Adapter by an association in the ObjectAdapter variant), which are the most important roles of the Adapter pattern. The number of unique occurrences in the gold standard, taking into account only those roles, was 45 (broken down in the second column of Table 6-2). Details of all the occurrences in the gold standard and those that were detected are provided in Appendix E.

6.3.2 Model Detail Experiments

The first set of experiments assessed the impact of different levels of model detail (i.e., behavioural specification detail) on accuracy when accounting for the official GoF variants only. Table 6-2 shows the data collected for these experiments. The acronyms used stand for the following: T (total number of occurrences), P (precision), R (recall) and U (undefined).

When the model only included the class diagram, detection had generally low precision (29%). The reason is that several patterns (e.g., Bridge and Strategy) did not have enough distinguishing structural features (unlike Singleton for example that required operation *getInstance* to be *static*). Conversely, for this reason, recall was relatively good (78%) as a lot of valid occurrences, including those of non-official variants, only differed behaviourally and thus were detected. However, non-official variant occurrences (i.e., occurrences of variants not described by GoF documentation) that sufficiently differed structurally were not detected (both Singleton and Observer had those, explaining their lower number of occurrences than in the GS).

Table 6-2 Data of initial GoF accuracy experiments with official variants ('U' means undefined)

Pattern	GS	Classes Only			Classes & Sequence			Classes & OCL			All Details		
	T	T	P	R	T	P	R	T	P	R	T	P	R
Abstract Factory	2	6	0.33	1.00	5	0.40	1.00	0	U	0.00	5	0.40	1.00
Factory Method	3	7	0.28	0.67	2	1.00	0.67	0	U	0.00	2	1.00	0.67
Singleton	4	2	1.00	0.50	0	U	0.00	2	1.00	0.50	2	1.00	0.50
Adapter	9	33	0.27	1.00	13	0.69	1.00	2	1.00	0.22	13	0.69	1.00
Bridge	5	23	0.17	0.80	8	0.50	0.80	0	U	0.00	8	0.50	0.80
Composite	2	3	0.67	1.00	2	1.00	1.00	1	1.00	0.50	2	1.00	1.00
Decorator	2	2	0.50	0.50	1	1.00	0.50	0	U	0.00	1	1.00	0.50
Chain Of Resp.	1	2	0.50	1.00	1	1.00	1.00	0	U	0.00	1	1.00	1.00
Command	6	27	0.22	1.00	10	0.60	1.00	0	U	0.00	10	0.60	1.00
Observer	5	1	1.00	0.20	1	1.00	0.20	0	U	0.00	1	1.00	0.20
Strategy	6	15	0.33	0.83	5	1.00	0.83	0	U	0.00	5	1.00	0.83
Overall	45	121	0.29	0.78	48	0.68	0.73	5	1.00	0.11	50	0.70	0.78

When the model included both class and sequence diagrams, we observed a radical improvement in precision (68%) as sequence diagrams allowed operation roles to be detected

more accurately. However, for patterns that did not have sufficiently distinguishable behaviour (e.g., Bridge only requires the Abstraction class to call an operation on the Implementation class, which is a behaviour that is hardly unique to this pattern), we detected invalid occurrences, which detracted precision. Recall slightly dipped though (73%) as none of the Singleton instances were detected because one of their behavioural conditions (looking for an assertion that the *getInstance* operation creates one instance only) could not be checked to hold by considering only sequence diagrams.

When the model included class diagrams and OCL constraints, we observed a dramatic drop in the number of detected occurrences across the board, as checking OCL constraints only was not sufficient to verify most behavioural conditions, especially for roles played by operations with side-effects (e.g., setter operations). This is because OCL, a side-effect free constraint language, cannot specify calls to such operations. (Notice that the way we specified the behavioural conditions, for example *localCall* in Figure 6-1, meant that as long as a UML operation had an OCL post-condition, then the analysis would be based on that, and would not default to *true* as in the case of class diagrams only.) This caused recall to fall to 11% and precision to have mixed results. Patterns with some detected occurrences, which are those with behavioural conditions that checked calls to some operations with no side-effects (e.g., the *getInstance* operation in Singleton or the *request* operation in Adapter) had high precision. Others, whose constraints always checked calls to operations with side effects (e.g., the *createProduct* operation in AbstractFactory) had no occurrences, hence an undefined precision.

When the model included all those details together, precision (70%) and recall (78%) were better (but not much) than when using class and sequence diagrams. This increase was due to

Singleton pattern occurrences becoming detectable thanks to the added OCL post-condition.

This also suggests that OCL post-conditions did not improve the results for the rest of the GoF patterns, since they cannot specify calls to operations with side effects used by those patterns.

We also observe that even when considering all those model details, perfect precision for some GoF patterns (e.g., Abstract Factory, Adapter, Bridge and Command) could not be achieved. The reason is the detection of occurrences that matched the pattern specifications while the design does not show intent to use the pattern. For example, some classes were reported as playing the *Adapter* role in the Adapter pattern, even though only a few (where it should have been many) of their operations were found playing the *request* role (this is a side effect of the specification matching one *request* operation at a time). Another example is that some classes were reported as playing the *Factory* role in the Abstract Factory pattern, since they were creating instances of some *Product* interface, but they were not really intended to be factories. Improving precision for these patterns further may require checking other criteria like: (i) metrics over the detected occurrences (e.g., the more Adapter occurrences are reported for a given class, each with a different *request* operation, the higher the chance the class really plays the *Adapter* role), or (ii) naming conventions (e.g., classes whose names end with *Factory* have a higher chance of actually playing the *Factory* role in the Abstract Factory pattern).

Similarly, we observe that perfect recall could not be achieved for a number of patterns (e.g., Factory Method and Singleton). This is due to differences between how these patterns are expected to be applied and how they have actually been applied in the model. For example, some classes that were playing the Singleton role in the Singleton pattern had a public *instance* variable and no *getInstance* operation, as opposed to a private *instance* and a public *getInstance*

operation as expected in the official variant. Improving recall in these instances further requires understanding how the patterns have actually been applied in the model (either from historical records or by manually inspecting a subset of the model) then accounting for the variability using one of the methods of Section 4.6. In the next section, we assess the impact of accounting for variability on accuracy (especially recall).

6.3.3 Pattern Variants Experiments

Recall that in the experiments discussed in Section 6.3.2 we only searched for the official variants of the GoF patterns. The second set of experiments was designed similarly (i.e., four sub-experiments based on model detail levels) except that we additionally assessed the impact of accounting for non-official variants that would be known to the pattern designer from previous projects (for instance). To simulate such situation in the case study, we asked the GMF architect² who designed the case study system to inspect one of the four packages (Model, Edit, Figure and UI) of the model to identify non-official variants of the chosen GoF patterns (i.e., ones not discussed in the official GoF description [Gamma et al., 1995]). We specified the ones that were found and we then used those specifications (in addition to the specifications of the official variants) when detecting patterns in the rest of the model (i.e., the three other packages). We repeated this for every package to remove any bias towards a particular package. The data of these experiments are shown in Table 6-3 in an aggregated form (i.e., for all patterns together).

We observed that precision slightly improved regardless of the level of details as compared to the precision obtained in Section 6.3.2: e.g., 72% in the all model details experiment compared to 70% as discussed in Section 6.3.2. This was due to the fact that the additional variants were specified based on their actual application in (one package of) the model, hence more model

structures were able to match pattern specifications. We also observed that recall improved further: e.g., 87% in the all model details experiment compared to 78% in Section 6.3.2. This was due to having more occurrences in the model that closely conformed to the newly added variant specifications. We also observed that both precision and recall were in the same range regardless of the specific package experiment, which suggests that it is enough to inspect a representative subset of the model. However, the definition of what a representative subset is may affect how much variability is discovered, and hence may impact the results. The more variants that can be accounted for, the better the results (especially recall) will be. For example, the fact that one architect defined the model and its packages may suggest some consistency across packages. However, if different architects developed different packages concurrently, one package may not necessarily be quite representative of other packages.

Table 6-3 Data of the subsequent GoF accuracy experiments with all known variants

Inspected Package	GS	Classes Only			Classes & Sequence			Classes & OCL			All Details		
	T	T	P	R	T	P	R	T	P	R	T	P	R
Model	45	127	0.32	0.91	51	0.70	0.80	7	1.00	0.16	55	0.73	0.89
Figure	45	128	0.39	0.93	52	0.71	0.82	7	1.00	0.16	56	0.73	0.91
Edit	45	127	0.32	0.91	51	0.71	0.80	5	1.00	0.11	55	0.73	0.89
UI	45	122	0.26	0.80	49	0.70	0.76	5	1.00	0.11	51	0.70	0.80
Overall	45	126	0.32	0.89	50	0.70	0.79	6	1.00	0.13	54	0.72	0.87

6.4 Evaluating Performance

The experiments of this case study were carried out on a laptop with 2.4 GHz core 2 duo processor and 3G of memory and running XP. We measured, using the Medini QVT tool, the average time (of five repetitions) taken to detect all patterns in each experiment excluding the time to load/unload the analyzed model.

We observed that the performance of detection was slowest (~35s) when the class diagram was considered only. At the other levels of model detail, performance was better due to behavioural conditions checking effectively pruning the search space. However, the experiment checking sequence diagrams was faster (~11s) than the one checking OCL constraints (~16s) as analyzing OCL constraints was more expensive (involved the cost of parsing the textual expressions into their AST).

When other variants were considered, detection slowed (~15% on average across levels) due to checking additional conditions. We also observed that using idioms as conditions enhanced performance by ~30% as it led to pruning the search space earlier (recall from Section 5.4 that condition pattern-uses reduce the candidate bindings for the pattern variables).

We also noticed that performance is a function of both model and pattern sizes. The more relevant elements exist in a model, the wider the search tree becomes, leading to more processing. Also, the more roles a pattern has, the deeper the search tree becomes, leading to more processing. However, the more conditions a pattern has, the more pruning occurs to the search tree, but also the costlier it is to check. Hence there is a trade-off between the effectiveness of using conditions for pruning and the cost of checking them.

6.5 Case Study Summary

The case study shows that our approach is applicable to the problem of detecting design patterns in MOF-based models. Specifically, the approach is adequate for concisely specifying (a representative subset of) the GoF family of design patterns with their structural conditions, behavioural conditions and variants. The use of idiom patterns improved the conciseness of most GoF patterns by an average of 68%. It also shows that the approach is capable of detecting

pattern occurrences with high accuracy, balancing both precision and recall. The best results (precision of 72% and recall of 87%) were achieved when a UML model contained appropriate details (class and sequence diagrams but not necessarily OCL constraints) and when known variants had been accounted for. We also suspect that checking other criteria, like metrics and naming conventions, in addition to inspecting a more representative subset of the model for variants may help boost the results even further. Results also suggest that the performance of the approach is reasonable relative to the size of the model. The best performance was achieved when enough relevant details existed in the model, which helped prune the search space faster.

Chapter 7: Detecting CF Patterns in BPMN—Case Study

The applicability of our approach was assessed in a second case study that involved the detection of a family of design patterns for a DSML (vs. UML). The assessment criteria (adequacy, accuracy and performance) we used in the first case study (Chapter 6) were also used in the second. However, the accuracy and performance experiments were simpler than in the first case study (i.e., we did not vary the model details or the considered variants). The main objective of this second case study was to confirm that our approach (of specifying patterns with VPML and detecting their occurrences in models with QVTR) was indeed generic and could also be used to specify and detect patterns of DSMLs with high accuracy and performance.

7.1 Case Study Design

The case study involved detecting a subset of the original Control Flow (CF) design patterns [Russel et al., 2006], which commonly arise in business process modeling, in a BPMN model. The subset consisted of ten (out of 20) of the CF patterns in three (out of five) categories: Basic Control Flow (Sequence, Parallel Split, Synchronization and Exclusive Choice), Advanced Branching and Synchronization (Multi Choice, Synchronizing Merge, Multi Merge and Discriminator) and Structural (Arbitrary Cycle and Implicit Termination). The choice of this subset was mainly influenced by the BPMN tool we used (i.e., RSA). The unconsidered patterns (mostly in the Multiple Instance and State-Based categories) required BPMN features that were

not supported by the tool. Nevertheless, we believe that the subset is representative of the CF family in general, where patterns involve chains of flow node elements connected by sequence flow relationships.

The analyzed BPMN model was an industrial model designed by IBM consultants as a pre-packaged solution for the financial services industry, defining common financial service processes. The model was created using RSA and was quite large with 91 process diagrams consisting of 500 activities, 274 gateways, 330 events and 550 sequence flows. Although we did not initially know whether the model contained occurrences of the CF patterns, we suspected this was the case as those patterns address very common problems in business process modeling and are often applied unintentionally.

We assessed adequacy by analyzing the ability of VPML to concisely specify the CF patterns, including their different conditions and official variants. Furthermore, we assessed accuracy of detection using the same metrics (precision and recall) that were used in the first case study. However, this time we did not assess the impact of varying the level of model detail. This is because, unlike UML models, BPMN process models include one kind of detail: the process flow elements. We also did not consider non-official variants because the CF patterns are simpler than GoF patterns and their variability is outlined in the official literature. Finally, we assessed the performance of detection similarly to the previous case study. Given the large size of the analyzed model, it was a good test bed for assessing the scalability of our detection.

7.2 Evaluating Adequacy

The chosen subset of CF patterns was specified in a VPML catalog (Appendix F). We discuss some interesting details of this catalog and assess its complexity.

7.2.1 Specification Details

When specifying the CF patterns, we observed that neither their generic descriptions [Russel et al., 2006] nor their BPMN-specific ones [Wohed et al., 2006; White, 2004] clearly identified the pattern roles: they mostly used examples that had obscurely named elements. Therefore, we had to define the roles ourselves, give them recognizable names and choose the ones to report on (i.e., the roles that help users with sufficient knowledge of the model and the pattern judge the validity of the occurrences without much further manual analysis). We found that the CF patterns mostly described chains of flow node elements connected by sequence flow relationships. After consulting with the model designers, we chose to report on nodes that started or ended such chains, since they helped define boundaries for the pattern occurrences. One exception is the Implicit Termination pattern, for which it was more interesting and concise to report on a process element that contains certain kinds of chains rather than on the individual chains.

Figure 7-1 Example of Sequence pattern (left) and its VPML pattern specification (right)

For example, consider the Sequence pattern (Figure 7-1 left), which exemplifies the Basic Control Flow category. It is defined as a sequence of uninterrupted activities of maximal length. (Although not in the original definition, we believed that requiring maximal length would likely give fewer but more useful occurrences.) We specified the Sequence pattern in VPML (Figure 7-1 right) with two roles: *start* activity and *end* activity, which designate the activities that start and end the sequence. We wanted to express that *start* can be connected to *end*, directly or

indirectly through intermediate activities, using unconditional sequence flows only. To do that, we defined a couple of contextual properties in the catalog (Figure 7-2). Property *nextUncondActivity* addresses the direct case by inspecting the outgoing sequence flows (of a given node), which do not have condition expressions, and selecting their target nodes (lines 1-2). Property *nextUncondActivityClosure* addresses the indirect case, by calculating the closure of the first property (lines 3-4), and was used to relate the *start* to the *end* roles (Figure 7-2). Next, to specify the maximal length aspect, we needed to restrict the *start* activity from having unconditional flows to previous activities. We did that by defining property *prevUncondActivity* on a node (line 5-6) and requiring it to be empty on the *start* role (Figure 7-2). Similarly, we required that *nextUncondActivity* be empty on the *end* role (Figure 7-2).

```

01 property FlowNode::nextUncondActivity: Set(Activity) = self.outgoing->
02   select(conditionExpression->isEmpty()).targetRef->select(oclIsKindOf(Activity));
03 property FlowNode::nextUncondActivityClosure: Set(Activity) =
04   self->closure(nextUncondActivity);
05 property FlowNode::prevUncondActivity: Set(Activity) = self.incoming->
06   select(conditionExpression->isEmpty()).sourceRef->select(oclIsKindOf(Activity));
07 property FlowNode::nextCondActivity: Set(Activity) = self.outgoing->
08   select(conditionExpression->notEmpty()).targetRef->select(oclIsKindOf(Activity));
09 property FlowNode::nextFlowNode: Set(FlowNode) =
10   self.outgoing.targetRef;

```

Figure 7-2 Specification of some contextual properties in the CF catalog

A number of CF patterns also incorporated gateways and other kinds of flow nodes in their chains. For example, consider the Exclusive Choice pattern that represents a decision point in a process where one of several branches is chosen. It features an exclusive gateway between a *before* and an *after* activities (Figure 7-3, left). In this case, we decided not to report the *gateway* role (notice the missing number box) as it could easily be figured out from the *before* and *after* role bindings. Also, we abstracted out the sequence flow relationships with contextual properties like *nextFlowNode* (Figure 7-2, lines 9-10), which provides the flow nodes connected to a given flow node by outgoing sequence flows.

Figure 7-3 Example of Exclusive Choice pattern (left) and its VPML pattern specification (right)

7.2.2 Assessing Specification Conciseness

The complete VPML specifications of the chosen subset of the CF patterns are provided in Appendix F. Table 7-1 shows some statistics collected from these specifications to help assess their conciseness. One observation is that the specifications have a small number of roles (1-4) and conditions (0-3), which is indicative of the simplicity of the CF patterns. Some patterns were also specified with multiple variants. For example, the Parallel Split pattern (Figure 7-4), which represents a split of a branch into multiple branches that can execute concurrently, was specified with two variants: a *before* activity splitting into many *after* activities using 1) a parallel gateway or 2) unconditional sequence flows. Some CF patterns also used idiom patterns. For example, the Discriminator pattern (Figure 7-5 right), which represents the convergence of N branches into one, used the parameterized N-out-of-M-Join idiom (Figure 7-5 left) setting its N parameter to 1.

Table 7-1 Statistics of the CF specifications in VPML

Category	Pattern	Reported Roles	Variants	Roles	Idioms	Conditions
Basic Control Flow	Sequence	start, end	1	2	0	3
	Parallel Split	before, after	2	3,2	0,0	3,2
	Synchronization	before, after	1	3	0	3
	Exclusive Choice	before, after	1	3	0	3
	Multi Choice	before, after	1	3	0	3
Advanced Branching & Synchronization	Synchronizing Merge	before, middle, after	1	4	1	3
	Multi Merge	before, after	2	3,2	0,0	3,2
	Discriminator	before, after	1	2	1	0
	Arbitrary Cycle	before, after	1	2	0	2
Structural	Implicit Termination	process	1	1	0	1

Figure 7-4 Specifications of the two variants of the Parallel Split pattern

Figure 7-5 Specification of N-out-of-M-Join idiom (left) and its usage by Discriminator (right)

7.3 Evaluating Accuracy

We carried out an experiment to assess the accuracy of detecting CF occurrences in the chosen model using precision and recall as metrics. The roles reported for each pattern (third column of Table 7-1) consisted mainly of activity roles (except for the structural pattern where there were other types of roles). Table 7-2 shows the data that was collected for the experiment. The gold standard had occurrences that were discovered by manual inspection of the model and by interviewing the consultants who defined the model. The number of unique occurrences in the gold standard, based on the reported roles, was 387 (second column of Table 7-2).

We observe that the results point to precision (88%) and recall (94%) values that are quite high. This is due to the nature of CF patterns: small and unambiguous, hence easier to detect accurately than GoF patterns in a UML model. There were a few reasons for not achieving a perfect accuracy. The first was that a number of sequence flows in the model had their conditions specified using annotations as opposed to condition expressions. This caused them to appear

unconditional, hurting the precision of some patterns looking for unconditional flows (e.g., Synchronization) and the recall of others looking for conditional flows (e.g., Exclusive Choice). The second reason is the use of some unknown pattern variants, which affected the recall of some patterns (e.g., Discriminator). The third reason was that some patterns (e.g., Multi Choice) did not have any occurrences causing their precision and recall to be undefined.

Table 7-2 Data of the CF detection experiment

Pattern	GS	Experiment			
	Total	Total	Precision	Recall	Time
Sequence	103	120	0.86	1.00	2.5s
Parallel Split	92	100	0.88	0.96	3s
Synchronization	62	69	0.90	1.00	3s
Exclusive Choice	20	15	1.00	0.75	1.5s
Multi Choice	0	0	U	U	1s
Synchronizing Merge	0	0	U	U	1s
Multi Merge	48	55	0.76	0.88	3s
Discriminator	5	4	1.00	0.80	0.5s
Arbitrary Cycle	23	17	1.00	0.74	1s
Implicit Termination	34	34	1.00	1.00	3s
Overall	387	415	0.88	0.94	19.5s

7.4 Evaluating Performance

The experiment was carried on a laptop with 2.4 GHz core 2 duo processor and 3G of memory and running XP. We measured, using Medini QVT, the average time (of 5 repetitions) taken to detect all patterns in the experiment excluding the time to load/unload the model.

We observed that the overall performance of detection (shown in the last column of Table 7-2) was reasonable (~19.5s) given the large size of the model. Patterns with more detected occurrences took longer to finish (~2-3s) than those with little or no occurrences (~1-1.5s) as they needed more processing. Also, more complex patterns that used idioms (e.g., Discriminator) benefited (~0.5s) from detecting their used idioms first, as they helped prune their search space.

7.5 Case Study Summary

The case study shows that our approach is also applicable to detecting a family of design patterns for a DSML. Specifically, the approach is adequate for concisely specifying the CF family of design patterns, on BPMN, with their constraints and variants. It also shows that the approach is capable of detecting pattern occurrences with high accuracy, both for precision (88%) and recall (94%). Results also suggest that the performance of the approach is reasonable (a matter of seconds) given the size of the analyzed model and specified patterns.

Chapter 8: Detecting Anti-Patterns in MOF—Case Study

In the previous two chapters, we reported on two large case studies where the applicability of our approach for detecting design patterns for UML and for a DSML was assessed. In this chapter, we report on a third case study where the applicability of the approach for detecting anti-patterns for a DSML was assessed. Recall that anti-patterns represent design smells or bad applications of design patterns. Early detection and fixing of anti-patterns in a model can be much cheaper than later on. We chose to study the anti-patterns of MOF because we noticed, through our involvement with OMG standards, that MOF-based metamodels tend to have a large number of anti-pattern occurrences (also called issues [OMG, 2012]). Some of these occurrences are symptoms of the complexity of designing a metamodel. Other occurrences are rather due to ambiguities in MOF (and its UML foundation), the lack of documented metamodeling idioms and best practices, the lack of formally-specified conventions or simply human error.

An obvious mitigation is to automate the checking of metamodels with tools. Unfortunately, a large number of metamodels are defined using UML tools as opposed to MOF tools (they only get converted to MOF in a post-processing step). The drawback is that most of these tools only implement constraints that are explicitly defined in the UML standard. They fail to implement other constraints that are informally implied by the UML semantics or those that are MOF-specific. In fact, this was one of the motivations for the MOF revision task force to drop MOF's

own metamodel in 2.4 in favour of using the UML metamodel directly (albeit with extra well-formedness constraints). This led us to research the MOF 2.4 metamodeling anti-patterns and define them in a catalog. We then formally specified the catalog in VPML and used it in a case study to detect anti-patterns in recent revisions of the UML metamodel.

The objective of the case study was to assess the applicability of our approach using two criteria: adequacy and performance. Notice that accuracy was not assessed for two reasons: (i) metamodels are purely structural and hence inherently less ambiguous than other models, making their anti-patterns inherently more precise; (ii) the recall of detection cannot be easily calculated due to the lack of a gold standard of anti-pattern occurrences for the UML metamodel. Instead, we chose to additionally assess the effectiveness of the new anti-pattern catalog itself, measured by the ratio of detected occurrences being recognized as issues and eliminated from a metamodel.

8.1 Catalog of Metamodeling Anti-Patterns

The catalog we defined consisted of 113 metamodeling anti-patterns in four categories (Appendix G.): (i) UML well-formedness (33 anti-patterns), (ii) MOF well-formedness (32 anti-patterns), (iii) semantic (33 anti-patterns), and (iv) convention (15 anti-patterns). In this section, we discuss each category and the strategy used for defining it. For brevity, we do not discuss all anti-patterns but focus on a subset, for which we detected occurrences in the case study.

8.1.1 UML Well-Formedness Anti-Patterns

As mentioned earlier, the MOF specification started using the UML metamodel as an abstract syntax in 2.4. Since UML is also used for general-purpose modeling, our first challenge was to

identify the subset of UML 2.4 that is relevant for MOF metamodeling and collect its well-formedness rules. First, we scanned the MOF 2.4 specification looking for the subset of UML (concrete) metaclasses and their (direct or inherited) properties that are relevant to MOF. Then, we compared this subset to the one being used in practice for defining two other metamodels (UML 2.3 and BPMN 2.0). We noticed some discrepancies that we discussed with the MOF 2.4.1 revision task force. For example, the UML metamodel used a Generalization element to specify inheritance between classes, while the MOF specification was still assuming a direct super class reference. Our final subset (shown in Table 8-1) was documented in the MOF 2.4.1 revision.

Table 8-1 The concrete UML metaclasses used for metamodeling with MOF 2.4.1

Association	Enumeration	LiteralInteger	PackageImport
Class	EnumerationLiteral	LiteralString	PackageMerge
Comment	Generalization	LiteralUnlimitedNatural	Parameter
Constraint	InstanceValue	OpaqueExpression	PrimitiveType
DataType	LiteralBoolean	Operation	Property
ElementImport	LiteralReal	Package	

Table 8-2 UML well-formedness anti-patterns (excerpt)

UML1	Class With Required Owner Property Defines Another Owner
UML2	Classifier Has Attribute Not Redefining Inherited One With Same Name
UML3	Comment Has No Annotated Elements
UML4	Constraint Expression Has Parse Errors
UML5	Constraint Has No Constrained Elements
UML6	Namespace Has Indistinguishable Members
UML7	Property Has Invalid Default Value
UML8	Property Is Derived But Has Default Value

Our next challenge was to collect the well-formedness constraints that are relevant to this subset of UML metaclasses. Some constraints were explicitly identified in the UML specification (e.g., circular generalizations are disallowed), while others had to be harvested from the described semantics (e.g., classes with required owner properties must not define other owner

properties). Based on those constraints, we defined 33 anti-patterns (violations of these constraints), some of which are shown in Table 8-2. An example (UML2) is a classifier with an attribute hiding (as opposed to redefining) a similarly named attribute in a general classifier. Another example (UML8) is a derived attribute with an explicit default value.

8.1.2 MOF Well-Formedness Anti-Patterns

The MOF 2.4 specification describes (sometimes implicitly) additional well-formedness constraints on the UML metamodel that should be checked when UML is used for metamodeling. This is because UML, being a general purpose modeling language, has looser semantics than expected by MOF. This loose semantics is tightened by additional MOF-specific well-formedness constraints. Based on these constraints, we defined 32 additional metamodeling anti-patterns, some of which are shown in Table 8-3. An example (MOF1) is a UML class flagged as active since being active has no meaning for a metaclass. Another example (MOF2) checks if a UML metaclass is allowed to be instantiated in a metamodel. Notice that this set of anti-patterns was defined for the Complete-MOF (CMOF) variant of MOF since it was more relevant to our case study (UML is a CMOF model). More anti-patterns will be needed for the Essential-MOF (EMOF) variant of MOF since it has even tighter semantics than CMOF.

Table 8-3 MOF well-formedness anti-patterns (excerpt)

MOF1	Association Does Not Have Two Member Ends
MOF2	Element Is Not Allowed In Metamodel
MOF3	Enumeration Has Operations
MOF4	Multiplicity Element Is Multi Valued But Has Default Value
MOF5	Named Element Has No Name
MOF6	Named Element Is Not Public
MOF7	Parameter Has Effect
MOF8	Typed Element Has No Type

8.1.3 Semantic Anti-Patterns

This category of anti-patterns defines UML designs that are well-formed syntactically but that could be problematic semantically when used for metamodeling. The set of 33 semantic anti-patterns were defined based on experience defining metamodels over the years. Table 8-4 shows a subset of those anti-patterns. An example (SEM6) is a classifier having redundant generalizations that are already implied by other generalizations leading to unnecessarily added complexity. Another example (SEM14) is an operation that is not flagged as a query (i.e., has no side effects on the model). Since operations are typically needed to facilitate writing OCL constraints in a metamodel, they are expected to be defined as queries. Another example (SEM20) is a property that is required but has no default value, forcing users to specify a value for it every time it is used.

Table 8-4 Semantic anti-patterns (excerpt)

SEM1	Association Has Asymmetric Redefinition
SEM2	Association Has Asymmetric Subsetting
SEM3	Association Is Bidirectional With Asymmetric Derived Ends
SEM4	Association IsDerived Conflicts With Ends IsDerived
SEM5	Classifier Has Ambiguous Non-Owned End
SEM6	Classifier Has Redundant Generalizations
SEM7	Classifier Is Abstract With One Direct Subtype
SEM8	Constraint Has Trivial Expression
SEM9	Constraint References Non Context Element Only
SEM10	Multiplicity Element Has Redundant Lower Bound
SEM11	Multiplicity Element Has Redundant Upper Bound
SEM12	Namespace Has Identical Constraints
SEM13	Operation Could Be Converted To Derived Attribute
SEM14	Operation Is Not Query
SEM15	Property Has Different Name Than Redefined Property
SEM16	Property Has Redundant Subsetting
SEM17	Property Is Composite And Typed By Data Type
SEM18	Property IsDerived Conflicts With IsReadOnly
SEM19	Property Is Optional With Default Value
SEM20	Property Is Required With No Default Value

8.1.4 Convention Anti-Patterns

This category defines anti-patterns that are considered violations to commonly used metamodel conventions. Such conventions may explicitly be documented in modeling specifications (e.g., the UML specification documents conventions that are used in defining its metamodel) or they might be implicitly applied. We have collected 15 such conventions and specified their violations as anti-patterns. Table 8-5 shows a subset of those anti-patterns. Some of them (CON2/3/6/7/8/9) are violations to naming conventions. Others (CON4/5) are violations to documentation conventions. Yet others (CON1/10/11/12) tighten some loose UML semantics like requiring the ordered set of member ends of associations to be used in a particular order (CON1).

Table 8-5 Convention anti-patterns (except)

CON1	Association Member Ends Are Reversed
CON2	Association Has Non-Default Name
CON3	Classifier Name Is Part Of General Classifier Name
CON4	Named Element Has No Documentation When It Should
CON5	Named Element Has Multiple Documentations
CON6	Named Element Is Not Alphabetic
CON7	Named Element Starts With Upper Case
CON8	Operation Has Return Parameter Not Named “result”
CON9	Property Is Boolean But Does Not Start With “is”
CON10	Property Is Derived With No Derivation Constraint
CON11	Property Derivation Constraint Does Not Reference Property
CON12	Typed Element Has Default Value Literal With Type Set

8.2 Case Study Design

The case study involved specifying the metamodeling anti-patterns (Section 8.1) using VPML and detecting them with a corresponding QVTR transformation in recent revisions (2.2, 2.3 and 2.4) of the UML metamodel. The objective was to assess the applicability of our approach for detecting anti-patterns of a DSML (MOF in this case). Specifically, we assessed (i) the adequacy to specify such complex anti-patterns, (ii) the performance of the detection and (iii) the

effectiveness of the anti-pattern catalog in detecting relevant problems. In order to evaluate the effectiveness of the detection, we used the expertise of the UML revision task force to review and judge the detected occurrences. In fact, the approach was used in realistic conditions to actually improve the UML 2.4 metamodel. Moreover, the UML metamodel with about 680 classes, 623 properties and 128 operations could be considered complex and representative of real-life metamodels, and hence serviced as a good test bed for performance.

8.3 Evaluating Adequacy

We specified the entire catalog of metamodeling anti-patterns in VPML. However, due to the size of the catalog (113 anti-patterns), we do not provide it here, but instead provide the corresponding QVTR transformation in Appendix G: it was more convenient to copy/paste the transformation text than capture and layout images of VPML specifications. However, we show both specifications for an example anti-pattern here (and more in Appendix G) to allow examining how they map to each other.

For example, consider the VPML specification of the UML anti-pattern `AssociationHasBothEndsComposite` in Figure 8-1. It specifies an association that has both of its member ends as composite, indicating that an element related by such an association to another element both contains it and is contained by it at the same time; an obvious contradiction. The specification defines three roles: an exposed role for the association itself and two non-exposed roles for the association member ends (recall that the decision to expose a role or not depends on whether the role is important when the pattern is used by other patterns). Both ends are constrained to be composite and different from each other.

The QVTR relation that detects this anti-pattern is shown in Figure 8-2. It defines one *checkonly* domain variable named *association* (line 2), with two interrelationships to its end role variables, *end1* and *end2* (lines 3-4), which are defined with their *isComposite* condition in the *when* clause (lines 14-15), along with a condition that they are not equal (line 16). The relation also defines an *enforce* domain to report all three roles (lines 6-12).

Figure 8-1 The VPML specification of anti-pattern AssociationHasBothEndsComposite

```

01  top relation AssociationHasBothEndsComposite {
02    checkonly domain in association:Association {
03      memberEnd = end1:Property {},
04      memberEnd = end2:Property {},
05    }
06    enforce domain out c:CatalogResult { name="UML Anti-Patterns",
07      pattern = p:PatternResult { name="AssociationHas...",
08        root = r1:RoleResult { name="association", element=association,
09          child = r2:RoleResult { name="end1", element=end1,
10            child = r3:RoleResult { name="end2", element=end2,
11              }}}}};
12  }
13  when {
14    end1:Property { isComposite = true }
15    end2:Property { isComposite = true }
16    end1 <> end2;
17  }
18 }

```

Figure 8-2 The QVTR specification of anti-pattern AssociationHasBothEndsComposite

Furthermore, in order to assess the overall adequacy of VPML to specify this large set of anti-patterns, we collected statistics from the catalog (broken down by category) in Table 8-6. Some anti-patterns had variants; hence we ended up with slightly more VPML patterns in some

categories, for a total of 122 pattern specifications (from the initial 113). In total, the catalog specified the anti-patterns with 207 roles (1-6 roles per anti-pattern). We observed that UML and semantic anti-patterns had an average of 2 roles, while MOF and convention anti-patterns had an average of 1.5 roles. We also observed that in total the catalog specified 324 conditions (1-11 conditions per anti-pattern) with an average of 2.86 conditions per anti-pattern (i.e., 1.56 conditions per role). Conditions varied in their conciseness with 64% expressed as attribute values and 36% expressed with OCL expressions. Furthermore, 20% of the conditions were further simplified by thirteen new operations and sixteen new derived properties that we added to the catalogs. The above analysis suggests that our approach was adequate to specify the catalog of anti-patterns concisely.

Table 8-6 Metrics of the metamodeling catalog specified with VPML

Category (#)	Patterns	Roles		Conditions			
		Avg.	Tot	Avg.	Tot.	Complex	Simplified
UML (33)	37	2.09	69	3.39	112	37	40
MOF (32)	32	1.56	50	1.59	51	22	3
Semantic (33)	36	2.03	67	3.18	105	34	20
Convention (15)	17	1.4	21	3.73	56	24	1

8.4 Evaluating Performance

The experiment of this case study was carried on a laptop with 2.4 GHz core 2 duo processor and 3G of memory and running XP. We measured, using Medini QVT, the average time (of 5 repetitions) taken to detect all anti-patterns (Table 8-7). Results indicate that it takes under a minute to run the whole catalog, which is efficient and reasonable to repeat frequently as the analyzed model is evolving. We note that the UML category takes particularly longer to finish due to anti-pattern UML4, which parses OCL constraints verifying their syntax.

Table 8-7 Average time in seconds for running metamodeling anti-pattern categories

UML Wellformedness	MOF Wellformedness	Semantic	Convention
22	8	15	5

8.5 Evaluating Effectiveness

We used the metamodeling anti-pattern specifications to check recent revisions (2.2, 2.3 and 2.4) of the standard UML metamodel. We obtained those revisions from OMG as UML models (recall that MOF 2.4-based metamodels are specified using UML). For each revision, we ran the QVTR transformation and detected many anti-pattern occurrences (2558 in 2.2, 2120 in 2.3, and 786 in 2.4). A complete report on those occurrences is provided in Appendix H. Notice that for the 2.4 revision, we ran the detection twice, once on a beta release and once on the final release. In the beta release, we detected 1671 occurrences (omitted from Table 8-8 for brevity but provided in detail in Appendix H), for which we logged issues to the UML 2.4 RTF along with proposed resolutions. Many of these resolutions made their way to the final release resulting in a much more solid and consistent revision for UML.

Furthermore, Table 8-8 shows the kind and the number of detected anti-pattern occurrences in each revision of UML. Our first observation is that the quality of the UML metamodel has been improving over time, which is expected given the mandate of the UML RTF to address issues with the metamodel. Specifically, the total number of anti-pattern occurrences has decreased by 17% from 2.2 to 2.3 and by 63% from 2.3 to 2.4 (final). When we checked the beta release of 2.4, we observed a 21% reduction from 2.3 but, more importantly, a 53% reduction between the beta and final releases of 2.4. Given that most of the metamodel changes between these 2.4 releases were to address issues raised by this case study, it shows the usefulness of automated model verification, and more specifically, the effectiveness of our approach in

realistic conditions, where a standard metamodel is being revised by an official task force (verifying the accuracy of the results).

Table 8-8 Total number of anti-pattern occurrences in three revisions of UML

Prob.	2.2	2.3	2.4	Prob.	2.2	2.3	2.4	Prob.	2.2	2.3	2.4
UML1	5	58	0	SEM1	23	25	0	SEM17	6	6	0
UML2	1	1	0	SEM2	208	203	0	SEM18	21	23	11
UML3	0	7	0	SEM3	6	6	6	SEM19	4	4	2
UML4	200	190	185	SEM4	37	38	0	SEM20	0	0	1
UML5	3	3	0	SEM5	1	160	151	CON1	43	0	0
UML6	12	0	0	SEM6	1	0	0	CON2	306	10	9
UML7	3	3	0	SEM7	4	4	4	CON3	1	1	1
UML8	14	14	14	SEM8	207	208	232	CON4	7	10	6
MOF1	1	0	0	SEM9	0	1	0	CON5	58	58	62
MOF2	1	1	0	SEM10	179	186	0	CON6	5	5	5
MOF3	1	1	0	SEM11	478	483	0	CON7	5	5	6
MOF4	2	2	0	SEM12	4	4	0	CON8	122	126	0
MOF5	443	141	0	SEM13	53	55	58	CON9	7	7	7
MOF6	17	0	0	SEM14	3	0	0	CON10	19	19	0
MOF7	9	9	0	SEM15	4	8	24	CON11	9	11	0
MOF8	3	3	0	SEM16	0	0	2	CON12	22	22	0

In this case study, we evaluated the effectiveness of the anti-pattern catalog (Section 8.1) as measured by the ratio of detected occurrences getting resolved (between the two most recent revisions, i.e. 2.3 and 2.4). The resolution decision was made purely by the UML RTF. Effectiveness of different categories of the catalogue varied as follows: UML (28%), MOF (100%), semantic (65%) and convention (65%). MOF occurrences fared very well because their anti-patterns are mostly noncontroversial. UML occurrences, on the other hand, did not fare as well only because one of the UML anti-patterns (UML4: constraints have parse errors) had a relatively large number of occurrences that required significant effort to resolve. In that case, the RTF decided to defer the resolutions to a later revision, which skewed the ratio. Without that specific anti-pattern, the UML category would have had a more reasonable ratio of 84%. The

semantic and convention categories were moderately effective as some occurrences, although recognized as valid, were controversial to resolve given the potential side effects of the fixes.

More generally, some detected occurrences did not get resolved mainly for one of the following reasons: a) the RTF ran out of time and deferred them to a future revision (e.g., UML4/8, SEM8/16/18, CON4/5); b) the cost of fixing them now (e.g., on existing tool implementations) outweighed the value (e.g., SEM5/7/13/15, CON3/6/7/9); c) they were judged as exceptions to the rules and hence would be tolerated (e.g., SEM3/19/20, CON2). As an example of the latter, for some associations detected by CON2, i.e., associations with non-default names, enforcing naming conventions would have led to ambiguous names.

8.6 Case Study Summary

The results of the case study suggest that our approach is adequate when used for specifying a large and complex set of anti-patterns. It also shows that the approach has a good performance as it could execute the entire catalog of metamodeling anti-patterns on the UML metamodel in about one minute clearly demonstrating this is a scalable, practical technology for model verification. Finally, results also show that the newly proposed catalog is effective at finding valid occurrences in realistic metamodels as it led to a 53% reduction of the issues detected in the UML 2.4 revision (which were all verified and agreed upon by the UML RTF).

Chapter 9: Discussion

The three case studies (Chapter 6, Chapter 7, and Chapter 8) clearly show that the proposed approach of specifying patterns with VPML and detecting them with QVTR works on any MOF-based modeling language and for both design patterns and anti-patterns. While carrying the case studies, we gained valuable insights that we discuss in this chapter.

9.1 Benefits and Costs of the Approach

The first insight is about the benefits and costs of using the approach. Recall that the benefits of analyzing models by detecting design patterns include supporting model maintenance (understanding where design patterns have been applied can help preserve or evolve these applications when a model needs to change) and model comprehension (decomposing models into more abstract and highly cohesive design fragments can aid their understanding). Also recall that the benefits of analyzing models by detecting anti-patterns include avoiding common mistakes that can be expensive to fix later on.

However, the incurred costs of the approach are split between tool developers and language designers. For tool developers, the cost is incurred when developing the required tools. We already incurred this cost (at least on the Eclipse platform) by developing a VPML editor (using GMF), a VPML detection engine (using JET and Medini QVT) and a PResults viewer (using GMF). However, this cost is marginal considering that such tools can be reused as is for any

modeling language. For language designers (or expert practitioners) who have a deep understanding of a language's syntax and semantics, the cost is incurred when specifying the known design patterns of a language. This cost is justified by the fact that these patterns are widely used within some communities (industries, organizations or projects) and hence detecting them in models can benefit those communities. In fact, as domain-specific languages get easier to define, language designers may find themselves work on more than one language throughout their career, and hence would appreciate the consistency of our approach.

9.2 Relevance and Applicability of the Approach

The second insight is the relevance and applicability of our approach. While the thesis focuses on detecting design patterns and anti-patterns, thus implying applicability to design languages only (e.g., UML for software design, BPMN for process design, and MOF for metamodel design), the approach is also applicable to non-design DSMLs with recurring patterns in their models (e.g., in a Family Tree DSML, a Twin pattern describes people with the same parents and the same birthdates). Detecting such patterns can help analyze those models (e.g., doctors may want to find people with a certain pattern of family illness). Moreover, it is hard to imagine a modeling language, however small its community (industry, organization or project), not having some recurring patterns that can benefit from our approach (even if such patterns are not officially described in a library due to lack of a formalism). Moreover, some patterns are applicable to more than one MOF-based modeling language (e.g., the control flow patterns apply to both BPMN process and UML activity diagrams) and a consistent approach to the formalization of such patterns would be expected by the industry. Also, modeling languages are increasingly used together to specify different aspects of a given system (e.g., BPMN for the business side and

UML for the technical side) and as such it is not hard to imagine patterns that cross the boundary of a single language. Such patterns will need to be specified using a language-independent approach like ours.

9.3 Relation to Other Modeling Activities

The third insight is how pattern detection relates to other modeling activities like model query, verification, reporting and transformation. One thing to realize is that pattern detection is a sophisticated form of model query (looking for elements satisfying certain conditions) where the detected model elements are combined into groups and assigned to the roles they play within the groups. Moreover, model verification is about ensuring that a model satisfies some desirable design criteria. For example, a design objective could be that a UML model only has top-level (non-nested) classes that are related to each other with generalizations and that only have public operations and private attributes. Such criteria can be specified as a pattern and hence model verification can be defined as detecting whether a model matches its design criteria patterns.

Furthermore, model reporting is about querying an input model for information and formatting it as a report (a form of model-to-text transformation). The model query part can be done with pattern detection when the information is structured as a pattern. In fact, this can simplify model reporting by using the metamodel of the pattern results (e.g., PResults) as a simpler input metamodel than the original input metamodel (e.g., UML). Similarly, model-to-model transformation is about querying an input model for information and converting that information into an output model. Again, the information can be defined as patterns to detect in input models. Also the pattern result model can be used as an intermediate simpler model to start from for converting the information to the output model.

9.4 Pattern Specification Process

The fourth insight is realizing that specifying patterns with VPML is itself a process. Pattern specifications are incrementally refined until acceptable levels of both performance and accuracy are achieved. Such refinement is often performed on sample models containing exemplary occurrences of the pattern. Performance refinement involves adjusting the specifications in order to prune the search space as early as possible. This can be achieved by specifying fewer roles at the top of the search tree and by adding relevant conditions. Accuracy refinement involves initially disabling many conditions in the pattern specifications, helping the recall (and finding opportunities to complete partial pattern applications) then gradually enabling some back to remove false positives, helping the precision. This process is repeated until a good balance between recall and precision is achieved. Notice that reaching a high level of any of precision and recall without hurting the other is sometimes challenging. This is due to the possibility of unexpected variants being used and the possibility of important details being missing in the analyzed models. One way around that is to refine pattern specifications periodically with knowledge of how they were actually being used in the past.

9.5 Complexity of Target Metamodel

The fifth insight relates to how to deal with the complexity inherent in the target metamodel (e.g., the UML metamodel is notoriously known for being complex). Typically, a pattern designer needs to be intimately familiar with the target metamodel's details. A good way to abstract these details is to define reusable VPML library catalogs with facilities (e.g. contextual operations, contextual properties and idioms) that abstract out much of these details. These

facilities can greatly simplify pattern specifications, making them more readable and maintainable.

Chapter 10: Limitations and Future Works

Even though we show that our approach is a practical solution to the problem of specifying and detecting pattern occurrences in MOF-based models, we also highlight some of its current limitations that we could not resolve due to lack of time. We also discuss possible mitigations for these limitations that we plan to investigate in the future.

10.1 Pattern Specification Genericity

One of the strengths of our approach is its ability to specify patterns of any MOF-based modeling language. However, in some cases, patterns may be applicable to more than one language (e.g., process patterns apply to both UML activity diagrams and BPMN process diagrams). In those cases, it would be desirable to specify such patterns in a generic way that can be reused for other languages. We call such a feature *pattern specification genericity*, i.e., the ability to specify generic pattern templates that can be bound to different languages to produce concrete pattern specifications. Currently, VPML requires such patterns to be defined separately for each language, making it hard to keep them in sync. A possible mitigation to investigate in the future is to define a small metamodel representing the pattern domain (concepts, attributes and relationships), specify the patterns against it then map it to each target language. Such mapping would be used to generate a corresponding QVTR transformation for each target metamodel.

10.2 Profile-Based Patterns

The proposed approach allows detecting patterns for modeling languages that are defined with a MOF metamodel. However, some languages (e.g., SysML [SysML, 2012] and SoaML [SoaML, 2012]) are rather defined with a UML profile, which is a light-weight extension mechanism for UML. A profile is a UML model that allows defining stereotypes, with new attributes, which are applicable to specific UML metaclasses. An instance of a stereotype can be attached to a UML element that conforms to one of the stereotype's applicable metaclasses. Since a profile is not a metamodel per se, we need to investigate in the future how VPML can be used to specify pattern roles played by stereotypes (since a stereotype cannot be a type for a pattern role directly).

Fortunately, a profile is still a UML model and hence considered MOF-based and accessible with OCL. Therefore, a possible solution could be to specify a pattern role played by a stereotype as being typed by one of the UML metaclasses extended by the stereotype, check the stereotype application with an OCL condition on the role, and define contextual attributes in the metaclass with OCL expressions that access the stereotype attributes. Alternatively, a stereotype role can be modeled with two VPML roles: one that is typed by the UML Stereotype metaclass (to be bound to the stereotype instance) and one that is typed by the extended UML metaclass (to be bound to the stereotyped element). The two roles would be linked by an interrelationship that points to the stereotype's extension property. In this case, the stereotype attributes can directly be constrained using attribute value conditions or interrelationships.

10.3 Unknown Variant Detection

The proposed approach allows dealing with expected pattern variability, as discussed in Section 4.6. This being said, it cannot completely deal with unexpected pattern variability, which can

happen for instance when a pattern is not correctly applied, i.e., when its presumed occurrence only satisfies a subset of the pattern's constraints, leading to low recall. However, the results of the GoF case study showed that unexpected variability can be dealt with in practice by partially inspecting the model looking for unknown variants and detecting them in the rest of the model. This led to a recall of 87%, which was quite high. Nevertheless, we plan to look for ways to achieve that with less manual work. One idea is to assign a confidence ratio to each constraint in the pattern and report all candidate occurrences along with their overall confidence scores.

10.4 Pattern Corrections

As shown in the third case study (Chapter 8), the approach can be used for model verification by detecting anti-patterns (bad smells) in models. In a typical verification session, a user would first inspect the detected occurrences and then address those deemed worthy of correction. However, for a large number of such occurrences, it would be tedious and error-prone for a user to correct them manually. Another use case is analyzing models looking for opportunities where design patterns could have been applied. These opportunities can also be formulated as patterns to detect in models. Again, the required refactoring to apply the patterns would be done manually. Unfortunately, our approach currently does not offer a way to automate the correction of patterns. We plan to investigate in the future an extension to VPML that allows specifying pattern corrections. Such specification can then be used to generate a transformation from the PResults model to the originally analyzed model to do the changes required by the corrections. Some of the anticipated challenges include the possibility of alternative corrections to choose from and the possibility of introducing problems with the corrections. We plan to investigate this area further in the future.

10.5 Presentation of Results

Our approach allows for presenting the detected pattern occurrences as a tree of role bindings. A unique occurrence is a complete branch from a root to a leaf role binding. In the PResults editor (Section 5.5.1), we show the total number of unique occurrences below each node in the tree to help a user, browsing the results, follow the concentration of occurrences. However, the concentration might not be the only useful criteria for a user. Other criteria could be the confidence score in the occurrences (Section 10.3) or the relevance of the occurrence to some user-defined concern. It would be useful to allow a user to influence the way the occurrences are presented. One possible solution to investigate is to allow a user to specify for each pattern a formula expressed in OCL that can be used to sort out the results.

Chapter 11: Conclusion

Pattern detection is an important technology in the context of model-driven engineering. Detecting occurrences of known design patterns supports the activities of model comprehension and maintenance. Detecting anti-patterns supports the activities of model verification. Pattern detection needs to be automated as the process is very resource intensive and tedious (due to the complexity of both models and patterns) when done manually. We present a novel approach to specify the patterns of any MOF-based modeling language. The approach allows specifying patterns visually with a new DSML called VPML that addresses many common pattern specification concerns like complexity, variability, accuracy and performance. Patterns specified in VPML can be translated automatically to a corresponding standard QVTR transformation that runs on an input model, where elements playing pattern roles are identified, producing a result model where pattern occurrences are reported in a concise and structured fashion (using a new metamodel called PResults). The approach was also prototyped on the Eclipse platform.

We also report on two case studies where the approach was used to detect occurrences of popular families of design patterns in large models conforming to widely used modeling languages. In the first case study, eleven (out of 23) GoF design patterns were specified on UML and detected in a design model (defined with different levels of detail) of a large project containing 45 unique pattern occurrences. The case study showed that the approach is adequate

to specify such a complex set of design patterns. It also showed that the approach is capable of yielding high accuracy levels with a precision of 72% and a recall of 87% when the model has an appropriate level of detail and when known variants (from previous projects for instance) have been accounted for. The case study also showed that the approach performs well (less than a minute) given the model size and pattern complexity. In the second case study, ten (out of 20) original control flow (CF) design patterns were specified on BPMN (a DSML) and detected in a large BPMN industrial model containing 387 unique pattern occurrences. The case study also confirmed that the approach is adequate to specify these design patterns concisely and detect their occurrences with high accuracy (a precision of 88% and a recall of 94%) and high performance (less than 20s). Both case studies provided results that suggest that our approach is both effective and efficient to automatically detect design patterns in MOF-based models.

We also report on a third case study where we defined a catalog of 113 MOF anti-patterns classified into four categories: UML well-formedness, MOF well-formedness, semantic and convention. We then specified the catalog with VPML, which was able to concisely express it. Finally, we used the catalog to detect anti-pattern occurrences in recent revisions of the standard UML metamodel. A large number of occurrences were detected and analyzed by the UML RTF. Results show that the catalog is effective at finding real occurrences in realistic metamodels, as it led to a 53% reduction of the occurrences detected in the UML 2.4 revision (as opposed to the 2.3 revision) as many of them got resolved by the RTF. Finally, the case study shows that the approach has a good detection performance as it could execute the entire catalog of anti-patterns on the large UML metamodel in under one minute showing the potential for rerunning the detection frequently as models change.

Together, the three case studies helped answer our research questions. With regard to the expressiveness of the approach, we found VPML to be sufficient to concisely specify different design patterns (e.g., GoF and CF) and anti-patterns (e.g., metamodeling), with varying complexities (in terms of number of roles, constraints and variants), on three different standard MOF-based modeling languages (e.g., UML, BPMN and MOF). With regard to the accuracy of the approach, we found the automated detection to be highly accurate (in terms of both precision and recall) when applied to realistic models that are known to have a large number of pattern occurrences. With regard to the performance of the approach, we found the automated detection to be highly scalable when applied to large models that we selected for each of the case studies: the models we selected are probably among the largest one in the domain.

Based on the findings of these case studies, although additional experiments are needed to confirm (or not) our observations, we believe that our approach has a high chance of working on a large, if not any, family of patterns and for a large, if not any, MOF-based language. However, more case studies need to be carried in the future to help further validate these findings. We also believe that VPML has characteristics that make it familiar and easy to use by pattern practitioners. Nevertheless, we plan to validate the usability of VPML in a separate study.

Bibliography

- Accelele (2012). Accelele – transforming models into code. <http://www.eclipse.org/accelele/>
- Ambler, S. (2002). Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. John Wiley & Sons.
- Amyot, D., Farah, H., and Roy, J.-F. (2006). Evaluation of Development Tools for Domain-Specific Modeling Languages. LNCS, vol. 4320, pp. 183-197.
- Bayley, I., and Zhu, H. (2010). Formal Specification of the Variants and Behavioural Features of Design Patterns. J. of System and Software, vol. 83(2), pp. 209-22.
- Beyer, D., Noack, A., and Lewerentz, C. (2005). Efficient relational calculation for software analysis. IEEE TSE, vol. 31 (2), pp. 137–149.
- Birkner, M. (2007). Objected-Oriented Design Pattern Detection Using Static and Dynamic Analysis in Java Software. Master Thesis, Univ. of Applied Sciences Bonn-Rhein-Sieg, Germany.
- Booch, G. (2012). Handbook of Software Architecture. <http://www.handbookofsoftwarearchitecture.com>
- BPMN (2010). Business Process Model and Notation (BPMN) v2.0. <http://www.omg.org/spec/BPMN/2.0/>
- Correa, A., Werner, C., and Zaverucha, G. (2000). Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns. LNCS, vol. 1844, pp. 33-191.
- Dijkmana, R., Dumasb, M., and Ouyangc, C. (2008). Semantics and analysis of business process models in BPMN. Information and Software technology, vol. 50 (12), pp. 1281-1294.
- Dong, J., Lad, D. and Zhao, Y. (2007a). DP-Miner: Design Pattern Discovery Using Matrix. Proc. 14th Ann. IEEE Int'l Conf. Eng. Computer-Based Systems (ECBS '07), pp. 371-380.
- Dong, J., Zhao, Y., and Peng, T. (2007b). Architecture and design pattern discovery techniques-a review. Software Engineering Research and Practice, pp. 621–627.
- Eden, A. (2002). LePUS: A Visual Formalism for Object-Oriented Architectures. The 6th World Conference on Integrated Design and Process Technology, Pasadena, California.
- Elaasar, M., Briand, L. and Labiche, Y. (2006). A Metamodeling Approach to Pattern Specification. LNCS, vol. 4199, pp.484-498.

- Elaasar, M., Briand, L. and Labiche Y. (2010). Metamodeling Anti-Patterns. <https://sites.google.com/site/metamodelingantipatterns>
- Enckevort, T. (2009). Refactoring UML Models: Using OpenArchitectureWare to Measure UML Model Quality and Perform Pattern Matching on UML Models with OCL Queries. In Proc. of OOPSLA'09, pp. 635-646.
- Eppstein, D. (1995). Subgraph Isomorphism in Planar Graphs and Related Problems. Proc. Sixth Ann. Symp. Discrete Algorithms, pp. 632-640.
- Feng, T., Zhang, J., Wang, H., and Wang, X. (2004). Software Design Improvement through Anti-patterns Identification. In Proc. of ICSM '04, pp. 534.
- Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. 1st edition.
- France, R., Kim, D., Ghosh, S. and Song, E. (2004). A UML-Based Pattern Specification Technique. IEEE TSE, vol. 30(3), pp. 193-206.
- Fulop, L., Ferenc, R., and Gyimothy, T. (2008). Towards a Benchmark for Evaluating Design Pattern Miner Tools. In Proceedings of CSMR, pp. 143-152.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison.-Wiley.
- GEF (2012). Graphical Editing Framework (GED). <http://www.eclipse.org/gef/>
- GMF (2012). Graphical Modeling Framework (GMF). <http://www.eclipse.org/gmf/>
- Gueheneuc, Y., Sahraoui, H., and Zaidi, F. (2004). Fingerprinting Design Patterns. Proceedings of the 11th Working Conference on Reverse Engineering (WCRE), pp. 172-181.
- Gueheneuc, Y., and Antoniol, G. (2008). DeMIMA: A Multilayered Approach for Design Pattern Identification. IEEE Tran. Software Engineering, vol. 34(5), pp. 667 - 684.
- Huang, H., Zhang, S., Cao, J., and Duan, Y. (2005). A Practical Pattern Recovery Approach Based on Both Structural and Behavioural Analysis. J. System Software, vol. 75 (1-2), pp. 69-87.
- Khomh, F., Vaucher, S., Gueheneuc, Y., and Sahraoui, H. (2009). A Bayesian Approach for the Detection of Code and Design Smells. In Proc. of ICSQ '09, pp. 305-314.
- Kim, D., and Shen, W. (2008). Evaluating Pattern Conformance of UML Models: a Divide-and-Conquer Approach and Case Studies. Software Quality Journal, vol. 16 (3), pp. 329-359.
- Koenig, A. (1995). Patterns and Antipatterns. J. of OO Programming 8(1), pp. 46-48.
- Le Guennec, A., Sunye, G., and Jezequel, J. (2000). Precise modeling of design patterns. LNCS, vol.1939, pp. 482-496.
- M2M (2012). Model to Model (M2M). <http://www.eclipse.org/m2m/>
- M2T (2012). Model to Text (M2T). <http://www.eclipse.org/m2t/>

- Mak, J., Choy, C., and Lun, D. (2004). Precise modeling of design patterns in UML. Proc. of ICSE'04, pp. 252–261.
- Maplesden, D., Hosking, J., and Grundy, J. (2002). Design pattern modeling and instantiation using DPML. TOOLS Pacific, pp. 3–11.
- Marinescu, R. (2004). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proc. of ICSM '04, pp. 350-359.
- MDT (2012). Model Development Tools. <http://www.eclipse.org/modeling/mdt/>
- Medini (2012). Medini QVT. <http://projects.ikv.de/qvt>
- Meyer, M. (2006). Pattern-based Reengineering of Software Systems. In Proc. of WCRE '06, pp. 305-306.
- Mikkonen, T. (1998). Formalizing design patterns. In Proceedings of ICSE, pp.115–124.
- MOF (2006). Meta Object Facility (MOF) Core v2.0. <http://www.omg.org/spec/MOF/2.0/>
- Moha, N., Gueheneuc, Y., Duchien, L., and Le Meur, A. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. TSE, vol. 36(1), pp. 20-36.
- Munro, M. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In Proc. of the 11th Int'l Soft. Metrics Symposium, pp.15.
- OCL (2010). Object Constraint Language (OCL) v2.2. <http://www.omg.org/spec/OCL/2.2/>
- OMG (2012). OMG issues database. <http://www.omg.org/issues/>
- Papyrus (2012). Papyrus UML. <http://www.papyrusuml.org>
- Pettersson, N., and Lowe, W. (2006). Efficient and Accurate Software Pattern Detection. Proc. 13th Asia Pacific Software Eng. Conf., pp. 317-326.
- Pettersson, N., Lowe, W., and Nive, J. (2009). Evaluation of Accuracy in Design Pattern Occurrence Detection. IEEE TSE, vol. 36(4), pp. 575-590.
- QVT (2011). Query/View/Transformation (QVT) v1.1. <http://www.omg.org/spec/QVT/1.1/>
- RSA (2012). Rational Software Architect. <http://www.ibm.com/software/rational/products/swarchitect/>
- Russell, N., Hofstede, A., Aalst, W., and Mulyar, N (2006). Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22 , BPMcenter.org
- Seemann, J., and Gudenberg, J. (1998). Pattern-based design recovery of Java software. Proc. of the 6th Int'l symposium on Foundations of software engineering, pp. 10–16.
- SoaML (2012). Service Oriented Architecture Modeling Language 1.0. <http://www.omg.org/spec/SoaML/1.0/>
- Smith, J. (2005). SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code. Dissertation, Comp Sci. Dept., Univ. of North Carolina at Chapel Hill.

- Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). EMF: Eclipse Modeling Framework. 2nd edition.
- SysML (2012). OMG Systems Modeling language 1.3. <http://www.omg.org/spec/SysML/1.3/>
- Taibi, T., Check, D., and Ngo, L. (2003). Formal specification of design patterns-a balanced approach. *Journal of Obj. Tech.*, vol. 2 (4), pp. 127-140.
- Trcka, N., Aalst, W., and Sidorova, N. (2009). Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In *Proc. of CAISE '09, LNCS*, vol. 5565, pp.425.
- Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. (2006). Design Pattern Detection Using Similarity Scoring. *IEEE transaction on software engineering*, vol. 32(11), pp. 896-909.
- UML (2009). Unified Modeling Language (UML), Superstructure 2.3. <http://www.omg.org/spec/UML/2.3/>
- White, S. (2004). Process Modeling Notations and Workflow Patterns. In L. Fischer, editor, *Workflow Handbook 2004*, pages 265–294. Future Strategies Inc., Lighthouse Point, FL, USA.
- Wikipedia (2012): Adder (electronics). [http://en.wikipedia.org/wiki/Adder_\(electronics\)](http://en.wikipedia.org/wiki/Adder_(electronics))
- Wohed, P., Aalst, W., Dumas, M., Hofstede, A., and Russell, N. (2006). Pattern-based Analysis of BPMN - An extensive evaluation of the Control-flow, the Data and the Resource Perspectives (revised version). BPM Center Report BPM-06-17, BPMcenter.org

Appendix A: Complete VPML Metamodel

This appendix provides the complete VPML metamodel for quick reference in Figure A-11-1.

More details about the metamodel can be found in Chapter 3.

Figure A-11-1 The VPML metamodel for pattern definition

Appendix B: Summary of Mapping from VPML to QVTR

This appendix provides a summary of the mapping from VPML to QVTR in Table B-1. The first column represents a metaclass (or one of its properties) from the VPML metamodel. The second column represents the corresponding syntax in QVTR (using the names of some properties between angle brackets). More details about the mapping can be found in Section 5.3.

Table B-1 A summary of the mapping from VPML to QVTR

VPML Feature	QVTR Feature
Catalog	imports <imports>*; transformation <name> (in:<metamodel>*, out:PResults) extends <superClass>*;
EMOF::Property	property <name> : <type> = <derivation>;
ContextualProperty	property <context>::<name> : <type> = <derivation>;
EMOF::Operation	query <name> (<<parameter>>*) : <returnType> <{ <body> } >
ContextualOperation	query <context>::<name> (<<parameter>>*) : <returnType> <{ <body> } >
EMOF::Parameter	<name> : <type>
Pattern (isDependant=False)	top relation <name> {...}
Pattern (isDependant=True)	relation <name> {...}
Pattern::condition	when { <condition>* }
Pattern::basePattern	when { <basePattern.name> (<basePattern.role[isExposed=True]>*); }
Pattern::conditionPattern	when { <<usedPattern>> }
Pattern::corollaryPattern	where { <<usedPattern>> }
Pattern::reportedRole (i is in [1...n])	enforce domain out c:CatalogResult { name = "<catalog.name>", pattern = p:PatternResult { name="<name>", root = r1:RoleResult { name="<reportedRole1.name>", element=<reportedRole1.name>, <child = ri:RoleResult { name="reportedRolei.name, element=<reportedRolei.name>,>*
PatternUse	<usedPattern.name> (<<RoleBinding ParameterBinding>>*);

Parameter	primitive domain <name> : <type>;
Role (isExposed=True)	checkonly domain in <name> : <type> {...}
Role (isExposed=False)	when { <name> : <type> {...} }
Role::condition	role : Type {...} { <condition>* }
RoleAttribute	role : Type { <property> = <value> }
RoleInterrelationship	role : Type { <property> = <targetRole.name> : <targetRole.type> {} }
RoleBinding	<bindingRole>
ParameterBinding	<value>

Appendix C: Specification of GoF Patterns on UML

The Gang of Four (GoF) design patterns [Gamma, 1995] are commonly used for object-oriented software design. The patterns are defined into three categories: (i) creational, (ii) structural, and (iii) behavioural. In this appendix, we show a VPML catalog that specifies a subset of 11 (out of 23) GoF patterns from different categories. The catalog also specifies a set of idiom patterns that are used in the GoF specifications. For each pattern specification, we show both the VPML diagram and the corresponding UML diagram. Due to the large number of specifications, we do not show the details here. Instead, we publish them online at:

<https://sites.google.com/site/designpatternedetection/gof-specifications>

C.1 GoF Creational Patterns

Abstract Factory

Factory Method

Singleton

C.2 GoF Structural Patterns

Adapter

Bridge

Composite

Decorator

C.3 GoF Behavioural Patterns

Chain of Responsibility

Command

Observer

Strategy

C.4 GoF Idiom Patterns

Congolmeration

Creation

Delegation

Object Recursion

Redefinition

Appendix D: GMF Design Model

The Graphical Modeling Framework (GMF) [GMF, 2012] is an open-source project that provides a framework for building graphical modeling tools on Eclipse. The overall design of GMF follows closely the model-view-controller (MVC) architectural pattern [Booch, 2012] that is commonly used when designing user interfaces. In such architecture, the model layer, containing the business objects is separated from the view layer, the visual manifestation, by the controller layer that maps between them and keeps them in sync. The view and the controller parts of GMF are based on the Graphical Editing Framework (GEF) project [GEF, 2012].

The GMF design model was created by a GMF architect who modeled the structural aspects with class diagrams and the behavioural aspects with sequence diagrams and OCL constraints. Since the model is too large to show in this document, we publish it online at:

<https://sites.google.com/site/designpatternndetection/gmf-design-model>

Appendix E: GoF Pattern Occurrences in GMF

The GoF pattern specifications (Appendix C) were used to detect occurrences in the GMF design model (Appendix D). The following is a list of patterns whose occurrences were documented in the gold standard and other lists whose occurrences were detected in four experiments with varying levels of model detail and considering official variants only. In each case, the number of occurrences is shown between parentheses. Due to the large number of occurrences, we do not show the details here. Instead, we publish them online at:

<https://sites.google.com/site/designpatternndetection/gof-occurrences>

E.1 Gold Standard Occurrences

Abstract Factory (2)

Adapter (9)

Bridge (5)

Chain of Responsibility (1)

Command (6)

Composite (2)

Decorator (2)

Factory Method (3)

Observer (5)

Singleton (4)

Strategy (6)

E.2 Class Diagrams Only Occurrences

Abstract Factory (6)
Adapter (33)
Bridge (23)
Chain of Responsibility (2)
Command (27)
Composite (3)
Decorator (2)
Factory Method (7)
Observer (1)
Singleton (2)
Strategy (15)

E.3 Class and Sequence Diagrams Occurrences

Abstract Factory (5)
Adapter (13)
Bridge (8)
Chain of Responsibility (1)
Command (10)
Composite (2)
Decorator (1)
Factory Method (2)
Observer (1)
Strategy (5)

E.4 Class Diagrams and OCL Constraints Occurrences

Adapter (2)
Composite (1)
Singleton (2)

E.5 All Model Details Occurrences

Abstract Factory (5)

Adapter (13)

Bridge (8)

Chain of Responsibility (1)

Command (10)

Composite (2)

Decorator (1)

Factory Method (2)

Observer (1)

Singleton (2)

Strategy (5)

Appendix F: Specification of CF Patterns on BPMN

The Control Flow (CF) design patterns [Russel et al., 2006] are commonly used for business process design. The patterns are categorized into several categories including: (i) basic control flow, (ii) advanced branching and synchronization, and (iii) structural. In this appendix, we show a VPML catalog that specifies a subset of 10 (out of 20) CF patterns from different categories. The catalog also specifies a set of idiom patterns that are used in the CF specifications. For each pattern specification, we show both the VPML diagram and the corresponding BPMN. Due to the large number of specifications, we do not show the details here. Instead, we publish them online at: <https://sites.google.com/site/designpatternndetection/cf-specifications>

F.1 CF Basic Control Flow Patterns

Exclusive Choice

Parallel Split

Sequence

Synchronization

F.2 CF Advanced Branching and Synchronization Patterns

Discriminator

Multi Choice

Multi Merge

Synchronizing Merge

F.3 CF Structural PatternsArbitrary CycleImplicit Termination**F.4 CF Idiom Patterns**N-out-of-M Join

Appendix G: Specification of MOF Anti-Patterns

The metamodeling anti-patterns are problems that commonly arise when defining metamodels using MOF. The 113 anti-patterns are organized into four categories: (i) UML well-formedness (33), (ii) MOF well-formedness (32), (iii) semantic (33), and (iv) convention (15). In this appendix, we show the QVTR transformation that is generated automatically from the corresponding VPML catalog. Due to the large size of the transformation, we publish it online at: <https://sites.google.com/site/metamodelingantipatterns/catalog>. At the end of this Appendix, we also provide some examples where we show the VPML specifications and the corresponding QVTR relations for one anti-pattern in each category to help the reader examine how they map to each other.

G.1 UML Well-Formedness Anti-Patterns

Association Generalization With Inconsistent Ends

Association Has Both Ends Composite

Association Has Multivalued Container End

Association Owns Composite End

Classifier Has A Final Parent

Classifier Has Attribute Not Redefining Inherited One With Same Name

Classifier Has Generalization Cycle

Classifier Has Operation Not Redefining Inherited One With Same Signature

Classifier Is Concrete With Abstract Operation

Classifier Is Concrete With Unsatisfied Derived Union Lower Bound

Class Is Concrete With One Optional Id

Class With Required Owner Defines Another Owner

Comment Has No Annotated Elements

Constraint Has Invalid Expression

Constraint Has No Constrained Elements

Constraint Is Applied To Itself

Multiplicity Element Has Negative Lower Bound

Multiplicity Element Has Upper Small Than Lower Bound

Namespace Has Indistinguishable Members

Opaque Expression Has Unmatching Body And Language

Operation Has More Than One Return Parameter

Operation Is Inconsistent With Redefined Operation

Operation Is Query With No Return Parameter

Parameter Has Invalid Default Value

Property Has Invalid Default Value

Property Has Invalid Subsetting Context

Property Is Derived Union But Not Derived

Property Is Derived Union But Not Read Only

Property Is Derived With Default Value

Property Is Inconsistent With Redefined Property

Property Is Inconsistent With Subsetted Property

Redefinable Element Has Leaf Redefined Element

Redefined Element Has Invalid Redefinition Context

G.2 MOF Well-Formedness Anti-Patterns

Association Does Not Have Two Member Ends

Behavioural Feature Is Not Sequential

Class Has More Than One Id

Class Has Nested Classifier

Class Is Active

Constraint Specification Is Not Opaque Expression

Element Import Has Alias

Element Import Is Not Public

Element Is Not Allowed In Metamodel

Enumeration Has Attributes

Enumeration Has Operations

Enumeration Literal Has Specification

Feature Is Static

Generalization Is Not Substitutable

Multiplicity Element Is MultiValued With Default Value

Named Element Has No Name

Named Element Is Not Public

Opaque Expression Has No Or Empty Body

Operation Has Non Class Exception

Package Import Is Not Public

Parameter Has Effect

Parameter Has Invalid Default Value

Parameter Is Exception

Parameter Is Stream

Property Has Invalid Default Value

Property Has Qualifier

Property Has Shared Aggregation

Typed Element Has Association Type

Typed Element Has No Type

Typed Element Is Complex With Default Value

Typed Element Is Enumeration With Non Instance Value Default

<u>Typed Element Is Primitive With Non Literal Default</u>
--

G.3 Semantic Anti-Patterns

<u>Association Has Asymmetric Redefinition</u>
--

<u>Association Has Asymmetric Subsetting</u>
--

<u>Association Is Abstract</u>

<u>Association Is Bidirectional With Asymmetric Derived Ends</u>
--

<u>Association Is Derived Conflicts With Ends Is Derived</u>
--

<u>Association Owns Both Ends</u>

<u>Class Does Not Have Container Property</u>

<u>Classifier Has Ambiguous Non Owned Ends</u>
--

<u>Classifier Has Redundant Generalizations</u>

<u>Classifier Is Abstract With No Concrete Subtypes</u>

<u>Classifier Is Abstract With One Direct Subtype</u>

<u>Classifier Is Concrete With Single Weaker Subset Of Derived Union</u>
--

<u>Classifier Owns Attributes That Subset Redefined Attributes</u>
--

<u>Constraint Has Trivial Expression</u>
--

<u>Constraint References Non Context Element Only</u>

<u>Data Type Has Class Typed Attributes</u>

<u>Multiplicity Element Has Redundant Lower Bound</u>

<u>Multiplicity Element Has Redundant Upper Bound</u>

<u>Multiplicity Element Has Zero Upper Bound</u>
--

<u>Namespace Has Identical Rules</u>

<u>Operation Could Be Converted To Derived Attribute</u>
--

<u>Operation Has No Return Type</u>

<u>Operation Has Return Type But With No Body</u>

<u>Operation Is Abstract With Body Condition</u>
--

<u>Operation Is Not Query</u>

<u>Property Has Different Name Than Redefined Property</u>
--

Property Has Redundant Redefinition

Property Has Redundant Subsetting

Property Is Composite And Typed By Data Type

Property Is Derived Conflicts With Is Read Only

Property Is Optional With Default Value

Property Is Required With No Default Value

Property Type Conflicts With Association

G.4 Convention Anti-Patterns

Association Has Invalid Order Of Member Ends

Association Has Non Default Name

Classifier Name Is Part Of General Name

Comment References Non Owner

Named Element Has Multiple Documentations

Named Element Has No Documentation

Named Element Is Not Alphabetic

Named Element Starts With Lower Case

Named Element Starts With Upper Case

Operation Has Return Parameter Not Named Result

Parameter Is Boolean But Does Not Start With Is

Property Default Literal Has Type

Property Derivation Does Not Reference Property

Property Is Boolean But Does Not Start With Is

Property Is Derived With No Derivation

G.5 Examples

G.5.1 UML: Class With Required Owner Property Defines Another Owner

This anti-pattern detects a UML class that has (directly or through inheritance) two owner properties, one of which is required. This makes the other property impossible to satisfy (if also required), or useless (if optional). This excludes the case when the required property is a subset or a redefinition of the second property. The VPML specification of the anti-pattern is shown in Figure G-1. The corresponding QVTR relation is shown in Figure G-2. Some contextual properties used in the specifications are shown in Figure G-3.

Figure G-1 VPML specification of anti-pattern **ClassWithRequiredOwnerDefinesAnotherOwner**

```

01 top relation ClassWithRequiredOwnerDefinesAnotherOwner {
02   checkonly domain source Owned:Class {
03     associationEnd = owner1:Property {}
04     allAssociationEnds = owner2:Property {},
05   };
06   enforce domain target c:CatalogResult {name = 'UML Anti-Patterns',
07     pattern = p:PatternResult {name = 'ClassWithRequiredOwnerDefine...',
08       root = r1:RoleResult {name = 'Owned', element = Owned,
09         child = r2:RoleResult {name = 'owner1', element = owner1,
10           child = r3:RoleResult {name = 'owner2', element = owner2
11         }}}}};
12   when {
13     owner1:Property { lower = 1
14       } { otherEnd.aggregation = AggregationKind::composite };
15     owner2:Property {
16       } { otherEnd.aggregation = AggregationKind::composite };
17     owner1 <> owner2;
18     owner1.allSubsettedProperties->excludes(owner2);
19     owner1.allRedefinedProperties->excludes(owner2);
20   }}

```

Figure G-2 QVTR relation of anti-pattern ClassWithRequiredOwnerDefinesAnotherOwner

```

01 property Property::otherEnd : Property =
02   self.association.memberEnd->any(e | e <> self);
03 property Type::associationEnd : Set(Property) =
04   Property.allInstances()->collect(e | e.otherEnd.type = self);
05 property Classifier::allAssociationEnds : Set(Property) =
06   self.associationEnd->union(self.general.allAssociationEnds);
07 property Property::allSubsettedProperties : Set(Property) =
08   self->closure(subsettedProperty);
09 property Property::allRedefinedProperties : Set(Property) =
10   self->closure(redefinedProperty);

```

Figure G-3 QVTR relation of anti-pattern ClassWithRequiredOwnerDefinesAnotherOwner

G.5.2 MOF: Typed Element Is Primitive With Non-Literal Default

This anti-pattern detects a UML typed element (e.g., a property) that is typed by a primitive type but has a default value that is not a literal. The VPML specification of the anti-pattern is shown in Figure G-4. The corresponding QVTR relation is shown in Figure G-5.

Figure G-4 VPML specification of anti-pattern TypedElementIsPrimitiveWithNonLiteralDefault

```

01 top relation TypedElementIsPrimitiveWithNonLiteralDefault {
02   checkonly domain source typedElement:TypedElement {
03     defaultValue = default:ValueSpecification {}
04   } { type.ocIsKindOf(PrimitiveType) {} };
05   enforce domain target c:CatalogResult {name = 'MOF Anti-Patterns',
06     pattern = p:PatternResult {name = 'TypedElementIsPrimitiveWith...',
07       root = r1:RoleResult{name = 'typedElement',element = typedElement,
08         child = r2:RoleResult {name = 'default', element = default,
09       }}}};
10   when {
11     default:ValueSpecification {
12       } { not ocIsKindOf(LiteralSpecification) };
13   }
14 }

```

Figure G-5 QVTR relation of anti-pattern TypedElementIsPrimitiveWithNonLiteralDefault

G.5.3 Semantic: Association Has Asymmetric Redefinition

This anti-pattern detects ends of an association that do not symmetrically redefine ends of another association. This happens when the opposite of a redefining end does not redefine (or subset) the opposite of its redefined end. The VPML specification of the anti-pattern is shown in Figure G-6. The corresponding QVTR relation is shown in Figure G-7.

Figure G-6 VPML specification of anti-pattern AssociationHasAsymmetricRedefinition

```

01 top relation AssociationHasAsymmetricRedefinition {
02   checkonly domain source redefining1:Property {
03     otherEnd = redefining2:Property {},
04     redefinedProperty = redefined1:Property {}
05   };
06   checkonly domain source redefined1:Property {
07     otherEnd = redefined2:Property {}
08   };
09   enforce domain target c:CatalogResult{name = 'Semantic Anti-Patterns',
10     pattern = p:PatternResult {name = 'AssociationHasAsymmetricRed...',
11       root= r1:RoleResult {name='redefining1', element = redefining1,
12         child= r2:RoleResult {name='redefining2', element = redefining2,
13           child= r3:RoleResult {name='redefined1', element = redefined1,
14             child= r4:RoleResult {name='redefined2', element=redefined2
15         }}}}}};
16   when {
17     redefining2.redefinedProperty->excludes(redefined2);
18     redefining2.subsettedProperty->excludes(redefined2);
19   }
20 }

```

Figure G-7 QVTR relation of anti-pattern AssociationHasAsymmetricRedefinition

G.5.4 Convention: Operation Has Return Parameter Not Named Result

This anti-pattern detects an operation that has a return parameter that is not named 'result' (a convention used in defining metamodel operations that return a value). The VPML specification of the anti-pattern is shown in Figure G-8. The corresponding QVTR relation is shown in Figure G-9.

Figure G-8 VPML specification of anti-pattern OperationHasReturnParameterNotNamedResult

```

01 top relation OperationHasReturnParameterNotNamedResult {
02   checkonly domain source operation:Operation {
03     ownedParameter = return:Parameter {}
04   };
05   enforce domain target c:CategoryResult{name='Convention Anti-Pattern',
06     pattern = p:PatternResult {name = 'OperationHasReturnParameter...',
07       root = r1:RoleResult {name = 'operation', element = operation,
08         child = r2:RoleResult {name = 'return', element = return,
09       }}}};
10   when {
11     return:Parameter {
12       direction = ParameterDirectionKind::return
13     } { return.name <> 'result' };
14   }
15 }

```

Figure G-9 QVTR relation of anti-pattern OperationHasReturnParameterNotNamedResult

Appendix H: Anti-Pattern Occurrences in UML Metamodel

The metamodeling anti-pattern specifications (Appendix G) were used to detect occurrences in four revisions (2.2., 2.3, 2.4 beta, 2.4) of the standard UML metamodel. The following is a list of anti-patterns whose occurrences were detected in each revision along with the number of those occurrences between parentheses. The occurrences are classified by UML revision then category. Due to the large number of occurrences, we do not show the details here. Instead, we publish them online at: <https://sites.google.com/site/metamodelingantipatterns/casestudy>

H.1 UML 2.2 Anti-Pattern Occurrences

<u>UML Anti-Pattern Occurrences (238)</u> <ul style="list-style-type: none"> <u>Class Has Multiple Containers And At Least One Is Required (5)</u> <u>Classifier Has Attribute Not Redefining Inherited One With Same Name (1)</u> <u>Constraint Has Invalid Expression (200)</u> <u>Constraint Has No Constrained Elements (3)</u> <u>Namespace Has Indistinguishable Members (12)</u> <u>Property Has Invalid Default Value (3)</u> <u>Property Is Derived With Default Value (14)</u>
<u>MOF Anti-Pattern Occurrences (477)</u> <ul style="list-style-type: none"> <u>Association Does Not Have Two Member Ends (1)</u> <u>Element Is Not Allowed In Metamodel (1)</u>

<u>Enumeration Has Operations (1)</u> <u>Multiplicity Element Is Multi Valued With Default Value (2)</u> <u>Named Element Has No Name (443)</u> <u>Named Element Is Not Public (17)</u> <u>Parameter Has Effect (9)</u> <u>Typed Element Has No Type (3)</u>
<u>Semantic Anti-Pattern Occurrences (1255)</u> <u>Association Has Asymmetric Redefinition (23)</u> <u>Association Has Asymmetric Subsetting (208)</u> <u>Association Is Bidirectional With Asymmetric Derived Ends (6)</u> <u>Association Is Derived Conflicts With Ends Is Derived (37)</u> <u>Classifier Has Ambiguous Non Owned End (1)</u> <u>Classifier Has Redundant Generalizations (1)</u> <u>Classifier Is Abstract With One Direct Subtype (4)</u> <u>Constraint Has Trivial Expression (207)</u> <u>Multiplicity Element Has Redundant Lower Bound (179)</u> <u>Multiplicity Element Has Redundant Upper Bound (478)</u> <u>Namespace Has Identical Rules (4)</u> <u>Operation Could Be Converted To Derived Attribute (53)</u> <u>Operation Is Not Query (3)</u> <u>Property Has Different Name Than Redefined Property (4)</u> <u>Property Is Composite And Typed By Data Type (6)</u> <u>Property Is Derived Conflicts With Is Read Only (21)</u> <u>Property Is Optional With Default Value (4)</u>
<u>Convention Anti-Pattern Occurrences (604)</u> <u>Association Has Invalid Order Of Member Ends (43)</u> <u>Association Has Non Default Name (306)</u> <u>Classifier Name Is Part Of General Name (1)</u> <u>Element Has No Documentation (7)</u>

Named Element Has Multiple Documentations (58)
Named Element Is Not Alphabetic (5)
Named Element Starts With Upper Case (5)
Operation Has Return Parameter Not Named Result (122)
Property Default Literal Has Type (22)
Property Derivation Does Not Reference Property (9)
Property Is Boolean But Does Not Start With Is (7)
Property Is Derived With No Derivation (19)

H.2 UML 2.3 Anti-Pattern Occurrences

UML Anti-Pattern Occurrences (276)

Class Has Multiple Containers And At Least One Is Required (58)
Classifier Has Attribute Not Redefining Inherited One With Same Name (1)
Comment Has No Annotated Elements (7)
Constraint Has Invalid Expression (190)
Constraint Has No Constrained Elements (3)
Property Has Invalid Default Value (3)
Property Is Derived With Default Value (14)

MOF Anti-Pattern Occurrences (157)

Element Is Not Allowed In Metamodel (1)
Enumeration Has Operations (1)
Multiplicity Element Is Multi Valued With Default Value (2)
Named Element Has No Name (141)
Parameter Has Effect (9)
Typed Element Has No Type (3)

Semantic Anti-Pattern Occurrences (1431)

Association Has Asymmetric Redefinition (25)
Association Has Asymmetric Subsetting (203)

Association Is Bidirectional With Asymmetric Derived Ends (6)

Association Is Derived Conflicts With Ends Is Derived (38)

Classifier Has Ambiguous Non Owned Ends (160)

Classifier Is Abstract With One Direct Subtype (4)

Constraint Has Trivial Expression (208)

Constraint References Non Context Element Only (1)

Multiplicity Element Has Redundant Lower Bound (186)

Multiplicity Element Has Redundant Upper Bound (483)

Namespace Has Identical Rules (4)

Operation Could Be Converted To Derived Attribute (55)

Property Has Different Name Than Redefined Property (8)

Property Is Composite And Typed By Data Type (6)

Property Is Derived Conflicts With Is Read Only (23)

Property Is Optional With Default Value (4)

Convention Anti-Pattern Occurrences (275)

Association Has Non Default Name (10)

Classifier Name Is Part Of General Name (1)

Comment References Non Owner (1)

Element Has No Documentation (10)

Named Element Has Multiple Documentations (58)

Named Element Is Not Alphabetic (5)

Named Element Starts With Upper Case (5)

Operation Has Return Parameter Not Named Result (126)

Property Default Literal Has Type (22)

Property Derivation Does Not Reference Property (11)

Property Is Boolean But Does Not Start With Is (7)

Property Is Derived With No Derivation (19)

H.3 UML 2.4 Beta Anti-Pattern Occurrences

UML Anti-Pattern Occurrences (287)

Class Has Multiple Containers And At Least One Is Required (53)

Comment Has No Annotated Elements (7)

Constraint Has Invalid Expression (186)

Constraint Has No Constrained Elements (3)

Property Has Invalid Default Value (3)

Property Is Derived With Default Value (14)

Redefined Element Has Invalid Redefinition Context (21)

MOF Anti-Pattern Occurrences (30)

Enumeration Has Operations (1)

Multiplicity Element Is Multi Valued With Default Value (2)

Named Element Has No Name (15)

Parameter Has Effect (9)

Typed Element Has No Type (3)

Semantic Anti-Pattern Occurrences (1195)

Association Is Bidirectional With Asymmetric Derived Ends (6)

Association Is Derived Conflicts With Ends Is Derived (35)

Classifier Has Ambiguous Non Owned Ends (151)

Classifier Is Abstract With One Direct Subtype (4)

Constraint Has Trivial Expression (209)

Constraint References Non Context Element Only (1)

Multiplicity Element Has Redundant Lower Bound (183)

Multiplicity Element Has Redundant Upper Bound (481)

Namespace Has Identical Rules (4)

Operation Could Be Converted To Derived Attribute (58)

Operation Has Return Type But With No Body (1)

Operation Is Not Query (1)

<u>Property Has Different Name Than Redefined Property (23)</u> <u>Property Has Redundant Subsetting (2)</u> <u>Property Is Composite And Typed By Data Type (6)</u> <u>Property Is Derived Conflicts With Is Read Only (25)</u> <u>Property Is Optional With Default Value (4)</u> <u>Property Is Required With No Default Value (1)</u>
<u>Convention Anti-Pattern Occurrences (159)</u> <u>Association Has Non Default Name (12)</u> <u>Classifier Name Is Part Of General Name (1)</u> <u>Comment References Non Owner (1)</u> <u>Element Has No Documentation (13)</u> <u>Named Element Has Multiple Documentations (62)</u> <u>Named Element Is Not Alphabetic (5)</u> <u>Named Element Starts With Upper Case (6)</u> <u>Property Default Literal Has Type (22)</u> <u>Property Derivation Does Not Reference Property (11)</u> <u>Property Is Boolean But Does Not Start With Is (7)</u> <u>Property Is Derived With No Derivation (19)</u>

H.4 UML 2.4 Anti-Pattern Occurrences

<u>UML Anti-Pattern Occurrences (199)</u> <u>Constraint Has Invalid Expression (185)</u> <u>Property Is Derived With Default Value (14)</u>
<u>Semantic Anti-Pattern Occurrences (491)</u> <u>Association Is Bidirectional With Asymmetric Derived Ends (6)</u> <u>Classifier Has Ambiguous Non Owned Ends (151)</u> <u>Classifier Is Abstract With One Direct Subtype (4)</u> <u>Constraint Has Trivial Expression (232)</u> <u>Operation Could Be Converted To Derived Attribute (58)</u>

Property Has Different Name Than Redefined Property (24)

Property Has Redundant Subsetting (2)

Property Is Derived Conflicts With Is Read Only (11)

Property Is Optional With Default Value (2)

Property Is Required With No Default Value (1)

Convention Anti-Pattern Occurrences (96)

Association Has Non Default Name (9)

Classifier Name Is Part Of General Name (1)

Element Has No Documentation (6)

Named Element Has Multiple Documentations (62)

Named Element Is Not Alphabetic (5)

Named Element Starts With Upper Case (6)

Property Is Boolean But Does Not Start With Is (7)