ECMFA 2011, LNCS 6698, pp. 282-298, 2011.

Domain-Specific Model Verification with QVT

Maged Elaasar[1,3], Lionel Briand[2] and Yvan Labiche[3],

[1] IBM Canada Ltd, Rational Software, Ottawa Lab
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com
[2] Simula Research Laboratory & U. of Oslo,
Martin Linges v 17, Fornebu, P.O.Box 134, 1325 Lysaker, Norway
briand@simula.no
[3] Carleton University, Department of Systems and Computer Engineering
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
labiche@sce.carleton.ca

Abstract. Model verification is the process of checking models for known problems (or anti-patterns). We propose a new approach to declaratively specify and automatically detect problems in domain-specific models using QVT (Query/View/Transformation). Problems are specified with QVT-Relations transformations from models where elements involved in problems are identified, to result models where problem occurrences are reported in a structured and concise manner. The approach uses a standard formalism, applies generically to any MOF-based modeling language and has well-defined detection semantics. We apply the approach by defining a catalog of problems for a particular but important kind of models, namely metamodels. We report on a case study where we used the catalog to verify recent revisions of the UML metamodel. We detected many problem occurrences that we analyzed and helped resolve in the (latest) UML 2.4 revision. As a result, the metamodel was found to have improved dramatically by the experts defining it.

Keywords: Smell, Anti-Pattern, Specification, Detection, UML, MOF, QVT.

1 Introduction

Model-driven engineering (MDE) is a software methodology that is based on the use of models as a primary form of expression. In such methodology, models get defined and keep evolving continuously to cope with changing system requirements. Models are defined as instances of a metamodel, a higher-level model that describes the abstract syntax of a modeling language, which can either be general-purpose like UML [1] or domain-specific (DSML) like BPMN [2]. Metamodels are themselves defined using a DSML called the Meta Object Facility (MOF) [3] that is standardized by the Object Management Group (OMG). A MOF-based metamodel consists of a set of metaclasses, their attributes and relationships, plus constraints governing their integrity. Metamodel constraints are often specified using the Object Constraint Language (OCL) [4] that is based on first-order predicate logic and set semantics.

Model verification is an integral process of MDE that is concerned with checking models to find occurrences of known problems. Problems can be of different kinds: a) syntactic problems specified by the well-formedness constraints of metamodels and their extensions (e.g., UML profiles are extensions of UML); b) semantic problems describing poor design choices that are known to have a negative impact on some aspect (e.g., implementability, maintainability, usability, performance) of models; c) convention problems, which are violations to methodological, organizational or project-specific conventions (e.g., naming conventions).

Verifying (large) models manually is a time and resource consuming activity that is also error-prone (some problems are complex, cross-cutting many model elements). A better approach is to automate model verification. Such an approach should first allow problems to be specified declaratively (leading to concise and maintainable specifications) using a generic (i.e., adaptable to any DSML), flexible (i.e., supporting arbitrary, complex problems) and standard (i.e., familiar and portable) formalism. Second, it should also allow problems to be detected automatically (using their specifications) and directly (involving no data conversion) in models. Finally, it should allow problem occurrences to be reported in a concise (i.e., easy to inspect) and structured (i.e., showing all role bindings) manner. Several approaches ([14] to [24]) have been proposed in the literature. However, none of them satisfies all of the aforementioned requirements (more details in Section 2).

In this paper, we present three contributions. First, we propose adopting the pQVT approach, which has been used for design pattern specification and detection in [5], for model verification. Similar to a design pattern, a problem is composed of inter-related and constrained model elements playing unique roles in a given context. Only this time, the context is problematic and the detection leads to finding problem (vs. pattern) occurrences. We show how pQVT can be used to specify and detect arbitrary problems of any MOF-based DSML. Problems get specified with a QVT-Relations (QVTr) [6] transformation from input models (conforming to a MOF-based metamodel), where elements involved in problems are identified, to result models (conforming to the pResults metamodel [5]), where problem occurrences are reported in a structured and concise manner. pQVT uses a standard declarative formalism and provides powerful reuse semantics, allowing for modularizing problem specifications and handling of problem variants. Thanks to QVTr's well-defined execution semantics, problems are detected by simply running the transformations, producing concise result models containing any detected problem occurrences.

Second, we investigate the power of our approach by defining a catalog of problems for a specific DSML, namely MOF. We chose to study MOF as it is used to define many popular metamodels (e.g., UML and BPMN) that tend to have a large number of issues [7]. The catalog has 113 problems in different categories: syntactic (based on MOF well-formedness rules), semantic (based on metamodeling idioms and best practices) and convention (based on conventions used for standard metamodels).

Third, we report on a case study where we specified the catalog with pQVT. The approach was found to be very adequate for expressing such a large and complex catalog in a modular and concise manner. We then used the specification to detect problems with recent revisions (2.2, 2.3 and 2.4 beta) of the standard UML metamodel. We detected and analyzed hundreds of problem occurrences, reported them to the UML 2.4 revision task force (RTF), and helped resolve 53% of them in

ECMFA 2011, LNCS 6698, pp. 282-298, 2011. © Springer-Verlag Berlin Heidelberg 2011

the final UML 2.4 revision. We also assessed the performance of the catalog and found its detection scaling very well (finishing in under a minute), given the size of the catalog and the complexity of the UML metamodel.

The rest of this paper is structured as follows: Section 2 highlights related work; the detection of problems with pQVT is described in Section 3; Section 4 presents a catalog of problems for MOF-based metamodels; a case study where the catalog was used to verify the UML metamodel is discussed in Section 5; Section 6 enumerates the limitations and future works; finally, the conclusions are given in Section 7.

2 Related Work

In the literature of model verification, a problem is often described as one of two kinds. The first is a bad smell [8], which is a symptom that possibly indicates a deeper problem. The second is an anti-pattern [9], which is a bad solution to a recurring design problem (as opposed to a good pattern like in [10]). Over the years, many smells and anti-patterns got defined. For example, Fowler [8] provided 22 code smells and Brown et al. [11] described 40 anti-patterns (e.g., Blob). Language-specific problems have also been studied. Huzar et al. [12] overviewed consistency problems for UML. Koehler and Vanhatalo [13] specified process anti-patterns for BPMN.

Several approaches to specify and detect problems have been proposed in the literature. Travassos et al. [14] introduced manual reading techniques to detect code smells. However, manual approaches do not scale for large systems. Dhambri [15] presented a semi-automatic approach that was a compromise between automatic detection, which is efficient but loses track of context, and manual inspection that is slow and subjective. Still, semi-automatic approaches cannot scale with large models.

Marinescu [16] presented an automatic metric-based approach to detect deviations from good design patterns. Metric values were compared against thresholds. One problem here is that the choice of metrics and thresholds are always controversial. Munro [17] tried to address this by empirically justifying his choices.

Some approaches used logic systems. Trcka et al. [18] used temporal logic (CTL*) to formalize data-flow anti-patterns and used solvers to detect them. Correa et al. [19] encoded smells in Prolog and detected them using an inference engine. Unlike pQVT's ability to work on models directly, these approaches need pre-processing to convert data (e.g., models) first into suitable representations. In this case, they get converted into predicates so that solvers can operate on them. This conversion is expensive and makes live integration with modeling tools harder.

Moha et al. [20] proposed a rule-based DSL to specify smells with a way to generate detection algorithms from rules. They obtained good precision and recall. The work was later extended by Khomh et al. [21] to convert specifications into Bayesian Belief Networks, which allowed specifying probabilities on different rules, improving results as they get sorted based on confidence. However, their DSL only allowed checking pre-defined constructs and focused on structural aspects of software only. In contrast, pQVT leverages OCL and thus has a higher expressive power and uses a generic detection algorithm that works by interpreting problem specifications.

Graph-based techniques have also been used. Meyer [22] converted code into Abstract Syntax Graph (ASG) representation and specified anti-patterns as template ASG graphs to match. Feng et al. [23] represented code using an XML schema for software and defined anti-patterns as template XML documents to match. These techniques also involve converting data first into another representation before detection becomes possible. For pQVT, models are already graphs of model elements allowing QVTr transformations to process them directly without conversion.

OCL has also been used. Enckevort [24] defined rules for UML class diagrams using OCL constraints and used them to check models. One problem with OCL is the way problem occurrences are reported. Since constraints are written in the context of one metaclass, reporting is limited to elements of that metaclass violating the constraints. Other interesting elements involved in the problem cannot be reported on simultaneously. In contrast, pQVT produces result models with occurrences reporting all interesting roles. Also, as an extension to OCL, pQVT provides a more declarative syntax and complexity management features that simplifies problem specification.

In summary, approaches in previous works are not fully adequate or practical to specify arbitrary problems for MOF-based DSMLs and automate their detection. We believe our pQVT approach has a combination of capabilities that make it an adequate solution to this important problem. It is declarative, leverages a standard formalism (QVTr), applies consistently to any MOF-based DSML, has complexity management facilities, inspects models directly (no conversion needed), has well-defined detection semantics, uses an interpreted (vs. code-generated) detection algorithm, and produces strucured and concise result reports.

3 Problem Detection with pQVT

pQVT is a pattern specification and detection approach that was previously defined in [5]. In this paper we show how pQVT can be used for model verification as well. The approach is depicted in Figure 1. Modeling problems are specified with a QVTr transformation from an input model (conforming to any MOF-based metamodel), where model elements involved in problems are identified, to a result model (conforming to pResults, Section 3.5), where problem occurrences are reported in a concise and structured manner. In the remainder of this section, we use an example modeling problem to illustrate the process of problem specification and detection with pQVT. First, we define a general template for problem specification with pQVT. Then, we use it to gradually specify the details of a modeling problem, including specifying problem roles, a problem occurrence, and problem variants if any.

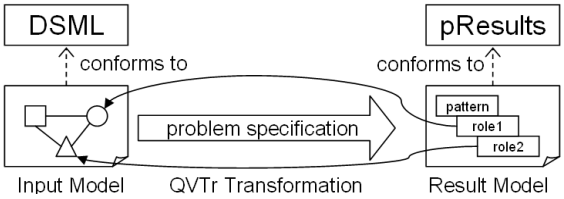

Figure 1 - Modeling problem detection with pQVT

3.1 Example Problem

The example problem is one of the UML well-formedness rules: a UML class should not define a new owner property when it already has a required one. The problem is depicted by the class diagram in Figure 2-left, where the class *Owned* is composed by two classes *Owner1* and *Owner2*, resulting in having two owner properties: *owner1* and *owner2*, respectively. According to UML semantics, an object can have a maximum of one owner reference at a time. Since *owner1* is required (has a multiplicity of 1), an *Owned* object must have reference to its owner through *owner1*, which makes *owner2* either impossible to satisfy (if it is required) or not useful (if it is optional). Figure 2-right shows a simplified subset of the UML metamodel that defines the concepts in the example problem.

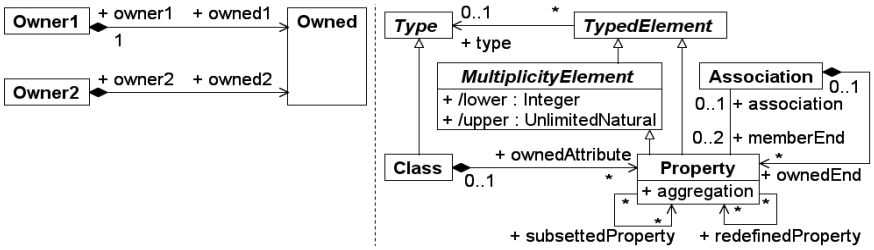

Figure 2 – The example problem (left) and a simplified subset of UML metamodel (right)

```
1 transformation Catalog (dsml:DSML, presults:pResults) {
2    top relation Problem {
3       checkonly domain dsml role:Type {/*problem role*/};
4       enforce domain presults c:Category {/*problem occurrence*/};
5       when {/*extra constraints*/}
6    }
7 }
```

Figure 3 – A template for problem specification using pQVT

3.2 Specification Template

A template for problem specification using pQVT is shown in Figure 3. A problem catalog is specified with a QVTr *transformation* (line 1) between two models: an input *dsml* model (conforming to some MOF-based DSML), where problems are detected, and an output *presults* model (conforming to pResults, Section 3.5), where problem occurrences are reported. Each problem in the catalog is specified with a *top relation* (line 2) declaring two kinds of variables: one or more *checkonly domain* variables (line 3) specifying the problem roles (their types and constraints) to detect in the *dsml* model, and a single *enforce domain* variable (line 4) specifying a problem occurrence to report in the *presults* model. A relation can optionally have a *when* clause (line 5) specifying extra constraints for the problem.

The semantics of problem detection with pQVT is based on the execution semantics of QVTr. When a transformation is executed, each *top relation* (specifying a problem) tries to find all combinations of elements in the input *dsml* model that satisfy the constraints of the *checkonly domain* variables (the problem roles) and those

in the *when* clause (extra problem constraints). For each such combination of elements, the *top relation* creates the elements specified by the *enforce domain* variable (the problem occurrence) in the *presults* model.

```
1  transformation Metamodeling (uml:UML, presults:pResults) {
2     top relation ClassWithRequiredOwnerDefinesAnotherOwner {
3        checkonly domain uml Owned:Class {
4           associationEnd = owner1:Property {},
5           associationEnd = owner2:Property {}
6        };
7        checkonly domain uml owner1:Property {
8           lower = 1
9        };
10       checkonly domain uml owner2:Property {};
11       when {
12          owner1 <> owner2;
13          owner1.otherEnd.aggregation = AggregationKind::composite;
14          owner2.otherEnd.aggregation = AggregationKind::composite;
15          owner2.subsettedProperty->excludes(owner1);
16       }
17    }
18    property Property::otherEnd : Property =
19       self.association.memberEnd->any(e | e <> self);
20    property Class::associationEnd : Set(Property) =
21       Property.allInstances()->collect(e | e.otherEnd.type = self);
22 }
```

Figure 4 – The specification of the example problem using pQVT

3.3 Problem Specification

Since the example problem is about the well-formedness of a UML model, a problem catalog is specified, in Figure 4, with a QVTr *transformation* (line 1) between an input *uml* model and an output *presults* model. The problem itself is defined with a *top relation* (line 2) within the catalog.

3.4 Role Specification

Identifying Roles. The problem description is used to identify the significant roles played by model elements in the problem. In the example problem, such roles are the *Owned* class and the *owner1* and *owner2* properties. The problem roles are therefore defined with *checkonly domain* variables (lines 3, 7 and 10) typed with corresponding metaclasses from UML (*Class* for *owner* and *Property* for *owner1* and *owner2*).

Adding Constraints. Adding more constraints to the roles enhances their precision. Simple constraints in the form of 'attribute=value' can be nested within the domain variable declarations. This is used to specify any expected values for roles' attributes (e.g., attribute *lower* of role *owner1* has a value of 1 to indicate it is required in line 8) or to specify role interrelations (e.g., properties *owner1* and *owner2* are related to class *Owned* through the attribute *associationEnd* in lines 4-5). Other constraints, more complex than simply 'attribute=value', are specified (in OCL) in the relation's *when* clause (e.g., a constraint requiring properties *owner1* and *owner2* to be distinct

in line 12; two constraints requiring them to be ends of composition associations in lines 13-14; and a constraint excluding the valid case of property *owner1* being a subset of property *owner2* in line 15). Some constraints may be complex or used several times in a transformation, in which case QVTr provides reuse facilities to simplify the transformation. One such facility is a *property* (an enhancement proposed in [5]) that is initialized with an OCL expression in the context of some DSML metaclass (e.g., *Property::otherEnd* in lines 18-19 gives the other member end across an association, and *Class::associationEnd* in lines 20-21 gives the properties accessible from a class over its associations). Those facilities can then be used in constraints across the transformation (e.g., lines 4, 5, 13, 14).

Relaxing Constraints. If a constraint is over restrictive, it may not get satisfied for some valid elements. Such constraint needs to be relaxed (removed or generalized). On the other hand, an overly loose constraint may get satisfied for some invalid elements. In practice, it takes some experimentation to reach an acceptable balance. While there is no generic way for generalizing constraints, role interrelationships may be generalized by making them transitive. For example, in Figure 5, three transitive properties are defined: *allAssociationEnds* (lines 12-13) allowing *owner1* to be an owner of class *Owned* or any of its super classes (line 3); *allSubsettedProperties* (lines 14-16) allowing *owner1* to be directly or transitively subsetted by *owner2* (line 8); and *allRedefinedProperties* (lines 17-19) allowing the exclusion of the valid case of *owner2* hiding *owner1* by directly or transitively redefining it (line 9).

```
1  top relation ClassWithRequiredOwnerDefinesAnotherOwner {
2     checkonly domain uml Owned:Class {
3        allAssociationEnds = owner1:Property {},
4        associationEnd = owner2:Property {}
5     };
6     ...
7     when {...
8        owner2.allSubsettedProperties->excludes(owner1);
9        owner2.allRedefinedProperties->excludes(owner1);
10    }
11 }
12 property Class::allAssociationEnds : Set(Property) =
13    self.associationEnd->union(self.superClass.allAssociationEnds);
14 property Property::allSubsettedProperties : Set(Property) =
15    self.subsettedProperty->union(
16       self.subsettedProperty.allSubsettedProperties);
17 property Property::allRedefinedProperties : Set(Property) =
18    self.allRedefinedProperty->union(
19       self.redefinedProperty.allRedefinedProperties);
```

Figure 5 – A more general specification of the example problem using pQVT

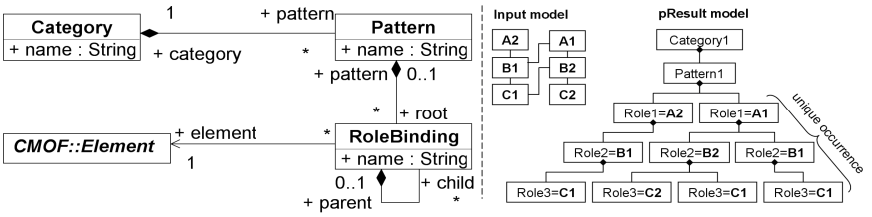

Figure 6 - pResults metamodel (left) and an example pResults model (right)

3.5 Problem Occurrence Specification

Problem detection may result in a set of problem occurrences. An occurrence is a unique mapping of roles to model elements playing those roles in a problem. The pQVT approach defines the pResults metamodel (Figure 6-left) to compactly represent occurrences of a given problem as a tree of *RoleBinding* objects under one problem (*Pattern*) object. At every level in the tree, role bindings map a unique role to a set of elements playing this role in an input model. A unique problem occurrence is a complete branch in the tree from a root to a leaf role binding. Related problem occurrences are grouped together under one *Category* object. Figure 6-right shows an example pResults model with occurrences pointing to elements in an input model.

In a pQVT problem specification, a pResuls problem occurrence is specified (Figure 7) as an *enforce domain* variable *c:Category* (line 1) that nests *p:Pattern* (line 2) that nests a *rb:RoleBinding* corresponding to each problem role (lines 3-5). The *element* attribute of each *RoleBinding* variable is assigned to the corresponding role's *checkonly domain* variable (e.g., *rb1*'s *element* in line 3 is assigned to variable *Owned* declared in line 3 of Figure 4).

```
1  enforce domain presults c:Category{ name='UML Problems',
2    pattern= p:Pattern{ name='ClassWithRequiredOwnerDefinesAnother',
3     root = rb1:RoleBinding{ name='Owned', element=Owned,
4      child = rb2:RoleBinding{ name='owner1', element=owner1,
5       child = rb3:RoleBinding{ name='owner2', element=owner2
6  }}}}};
```

Figure 7 – The specification of a problem occurrence for the example problem using pQVT

```
1  top relation ProblemWithVariants {
2     checkonly domain dsml role:Type {/*problem role*/};
3     when {/*extra conditions*/};
4     where { Variant1(role1, role2, ...);
5             Variant2(role2, role2, ...); }
6  }
7  relation Variant1 {
8     checkonly domain dsml role:Type {/*problem role*/};
9     enforce domain presults c:Category {/*problem occurrence*/};
10    when {/*extra constraints*/};
11 }
```

Figure 8 – Specification of a problem and its variants with pQVT

3.6 Variant Specification

Some problems may have variants (versions with slightly different roles and/or constraints). Specifying all variants ensures that all possible problem occurrences are detectable. However, specifying variants with separate *top relation*(s) is not efficient as it leads to duplication. Instead, a different problem specification template, shown in Figure 8, is used. In this template, the common roles (line 2) and constraints (line 3) are specified with a *top relation*, while the specific roles (line 8) and constraints (line 10) of each variant are specified with non-top *relation*(s) that get composed in the *top relation*'s *where* clause (line 4-5). A *where* clause extends a relation by composing other relations. Recall that a relation tries to find all combination of elements matching its *checkonly domain* variables. Now for each such combination, it calls the

ECMFA 2011, LNCS 6698, pp. 282-298, 2011. © Springer-Verlag Berlin Heidelberg 2011

composed relations in the *where* clause. Each call binds variables from a composing relation to the *domain* variables of a composed relation. This further constrains the common roles of a problem by the extra constraints of variants. Notice also that the *enforce domain* variable is moved to the variant relation (line 9) so that problem occurrences are only reported when all constraints for a variant have been satisfied.

4 Catalog of Metamodeling Problems

In the previous section, we showed how the pQVT approach can be used to specify problems of MOF-based DSMLs, using a UML problem as an example[1]. In this section, we present a catalog of problems we defined for another DSML, namely MOF itself. We chose to study MOF because we noticed, through our involvement with OMG standards, that MOF-based metamodels tend to have a large number of problem occurrences (called issues in [7]). Obviously, some of these occurrences are symptoms of the complexity of designing a metamodel, which requires expertise. However, many 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 use tool support to help with checking metamodels. Unfortunately, a large number of metamodels are defined using UML tools as opposed to MOF tools and only get converted to MOF as a post-processing step. The drawback is that most of these tools only implement constraints that are explicitly defined in the UML specification. They do not 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 dropping MOF's own metamodel in MOF 2.4 and using the UML metamodel directly, albeit with extra well-formedness constraints.

This led us to consider defining a catalog of MOF 2.4-based metamodeling problems. The catalog defines a total of 113 problems in four categories: UML well-formedness (33 problems), MOF well-formedness (32 problems), semantic (33 problems) and convention (15 problems). Well-formedness problems are based on constraints of the UML and MOF specifications. Semantic problems are based on metamodeling idioms and best practices. Convention problems are based on conventions used in developing standard metamodels. In the rest of this section, we elaborate on the strategy followed for defining each category. For brevity, we only show the subset of problems for which we actually detected occurrences in the case study in Section 5. A larger set of problems is provided online [26].

4.1 Well-Formedness Problems

UML Well-formedness. As mentioned above, MOF 2.4 uses the metamodel of UML 2.4, which is a general-purpose modeling language. Our first challenge was to identify the subset of the UML 2.4 metamodel that is relevant for metamodeling and collect its constraints. First, we identified a set of concrete UML metaclasses and their non-

[1] Approaches that work at the metamodel level apply equally to DSMLs and UML.

derived (direct and inherited) properties that have counterparts in MOF. We then compared it to the set that was actually used in defining two standard metamodels (UML 2.4 and BPMN 2.0) with UML. We noticed some differences that we had to reconcile. For example, UML used a Generalization element to specify inheritance between classes, while MOF used a direct super class reference. Finally, we validated the subset with key members of the MOF revision task force to ensure accuracy. Table 1 shows the metaclasses in this subset.

The next step was to collect the constraints that are relevant to this subset of metaclasses in UML. Some constraints were explicitly identified in the UML specification, while others had to be recovered from the described semantics. Based on those constraints, we defined 33 problems. Table 2 shows a subset of those problems. An example problem (UML2) is a classifier with an attribute hiding (as opposed to redefining) a similarly named one in a general classifier. Another example (UML8) is a property with an explicit default value even through it is derived.

Table 1 – The concrete UML metaclasses used for defining metamodels

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

Table 2 – UML well-fomedness problems (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

Table 3 – MOF well-fomedness problems (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

MOF Well-formedness. Some constraints are specific to MOF and we collected them from the MOF 2.4 specification [3]. Since the UML metaclasses used for metamodeling (Table 3) have more features and richer semantics than is needed for MOF, these constraints are meant to prevent usage of those features and semantics unrelated to MOF. We then defined 32 problems corresponding to those constraints.

Figure 3 shows a subset of those problems. An example (MOF1) is a class flagged as *active* since this has no meaning in a metamodel. Another example (MOF2) checks if a UML element is allowed to be in a metamodel. Notice that we only defined problems for the Complete-MOF (CMOF) variant since it is more relevant to the case study in Section 5. The Essential-MOF (EMOF) variant is more constrained and hence would need a bigger set of problems.

4.2 Semantic Problems

This category of problems defines situations that are well-formed according to the semantics of UML/MOF but could be problematic when implementing and/or using the metamodel. We collected 33 smells from experience defining metamodels over the years. Table 4 shows a subset of those smells. An example (SEM6) is a classifier having generalizations that are already implied by other generalizations, leading to redundancy. Another example (SEM14) is an operation not being flagged as a query (i.e., has no side effects). Operations are typically defined in a metamodel to facilitate querying models, especially by OCL expressions. Therefore, we need to check whether defining an operation as a non-query operation is really intended. Another example (SEM20) is a property that is required but has no default value, forcing modelers to specify a value for it every time in a model.

Table 4 – Semantic problems (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

4.3 Convention Problems

This category defines problems that are violations to common conventions adopted when defining metamodels. One may find these conventions specified explicitly as

part of metamodel specifications or one may find them implicitly applied. We have collected 15 such conventions and specified their violations as problems. Table 5 shows a subset of those problems. Some problems (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 association's member ends to be in a particular order.

Table 5 – Convention problems (excerpt)

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

5 Case Study

In this section, we report on a case study where we specified the catalog of metamodel problems presented above (Section 4) with pQVT (Section 3) and used it to verify recent revisions of the UML metamodel (defined with CMOF). The case study had three objectives. First, we wanted to assess the ability of pQVT to express a complex catalog of problems. Second, we wanted to assess the effectiveness of pQVT at detecting valid problem occurrences. The occurrences we detected in the standard UML metamodels were analyzed and results were shared with the UML Revision Task Force (RTF), which judged their validity. In fact, pQVT was used in realistic conditions to actually improve the UML 2.4 metamodel. Third, we wanted to evaluate the performance of pQVT on realistically large models to assess its scalability. The UML metamodel with about 680 classes, 623 properties and 128 operations, can be considered complex and representative of real-life metamodels.

5.1 pQVT Expressiveness

Due to the large number (113) of problems in the catalog (section 4), we cannot show all their pQVT specifications here. Instead, we collected some metrics in Table 6 to help the reader assess the effort involved. We chose to specify each category of problems in a separate QVTr transformation to make them more manageable. Each problem was specified using a *top relation* for a total of 122 relations (four problems had variants, thus needed some extra non-top relations). Problem specifications had a total of 207 roles with a range of 1-6 roles each. UML and semantic problems

involved more roles (~2) on average than MOF and convention problems (~1.5). The specifications also had a total of 324 constraints with a range of 1-11 constraints each. This means an average of 2.86 per problem and 1.56 per role, indicating that the specifications were generally concise. Constraints also varied in complexity with 64% specified using the form 'property=value' (i.e., nested in variable declarations) and 36% using other forms (i.e, in *when* clauses), indicating that the specifications were generally simple. Furthermore, 20% of constraints were simplified by using (13) queries and (16) derived properties that we defined in a reusable library and imported in the transformations. The above analysis suggests that pQVT had the expressive power and facilities needed to adequately specify such a large catalog of problems.

Table 6 – Metrics of the metamodeling catalog specified with pQVT

Category (Problems)	Relations	Roles		Constraints			
		Avg.	Total	Avg.	Total	In *when*	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

5.2 pQVT Effectiveness

We used the specified catalog to verify the (most recent) 2.2, 2.3 and 2.4 revisions of the standard UML metamodel. Recall that MOF 2.4 requires metamodels to be defined in UML. Therefore, we obtained those revisions from OMG as UML models. For each revision, we detected many problem occurrences: 2558 (2.2), 2120 (2.3) and 786 (2.4). A complete report is available online [26]. For 2.4, we first checked a beta revision and then based on our findings we reported problem occurrences (issues) to the UML RTF and helped resolve some of them. Finally, we checked the official 2.4 revision. Table 7 shows the number of occurrences of the identified problems. Our first observation is that the quality of the UML 2.x metamodel has been improving over revisions, which is expected given the mandate of the RTF to address issues with the metamodel. Specifically, the total number of problem occurrences has decreased by 17% from 2.2 to 2.3 and by 63% from 2.3 to 2.4. When we checked the beta revision of 2.4, we detected 1670 occurrences (omitted from Table 7 for brevity). This is a 21% reduction from 2.3 but, more importantly, a 53% reduction between the beta and official revisions of 2.4. Given that most of the metamodel changes between these two revisions (i.e., those in change ballot 11 [27]) were to address issues raised by this case study using pQVT, it shows the usefulness of automated model verification and more specifically, the effectiveness of pQVT in realistic conditions, where a standard metamodel is being revised by an official task force.

Nevertheless, different categories of the catalog varied according to the ratio of the detected occurrences getting resolved, as follows: UML (28%), MOF (100%), semantic (65%) and convention (65%). While MOF occurrences fared well given that they are not controversial, UML ones did not do as well because one of the problems (UML4: constraints have parse errors) had a relatively large number of occurrences that required significant effort to resolve. More generally, some occurrences did not

get resolved 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 tool migration) outweighed the value (e.g., SEM5/7/13/15, CON3/6/7/9); c) they were judged as exceptions to the rules (e.g., SEM3/19/20, CON2). An example of the latter is some associations detected in CON2 with non-default names, as the naming convention would have given them ambiguous names.

Table 7 – Total number of detected problem 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

5.3 pQVT Performance

We used the tool Medini-QVT [25] (with our performance tune-ups [5] like cashing query results) to specify and execute the QVTr transformations. We also used our pResults model viewer [5] to inspect and analyze problem occurrences. The detection was performed on a laptop with 2.4 GHz core 2 duo processor and 3G of memory running Windows XP. We recorded the average time for running each problem category on the UML metamodel (all revisions were in the same range). The times were as follows: UML (22s), MOF (8s), semantic (15s) and convention (5s). This means that it takes under a minute to run the whole catalog, which is very efficient and reasonable to repeat frequently as the analyzed model is evolving. We note that the UML category takes a bit longer due to problem UML4, which parses OCL expressions of constraints verifying their syntax.

6 Limitations and Future Work

The pQVT approach to model verification has some limitations and can still be improved further. For example, problem specifications could be made more portable, i.e. not tied to a particular DSML. We plan to resolve this issue by investigating

ECMFA 2011, LNCS 6698, pp. 282-298, 2011. © Springer-Verlag Berlin Heidelberg 2011

transformation genericity, where a generic DSML is defined for a problem domain and used to specify problems. Separate mappings can then be defined between such DSML and the real DSMLs. Another improvement could be to augment a problem specification with a way to auto-correct a problem occurrence. Another area to improve is the presentation of problem occurrences, which are currently not ordered. We plan to investigate ways to calculate importance scores for occurrences and order them accordingly, making inspection much more effective. Another possibility is to define a dedicated graphical pattern specification DSML whose models can be used to generate pQVT transformations along with all their boilerplate and idioms. Another possible work is to specify problems of other popular DSMLs, including UML profile-based ones, which could be interesting as some DSMLs are defined with UML profiles rather than MOF-based metamodels. Other case studies are also necessary to validate the flexibility and performance of the approach reported in this paper.

7 Conclusion

Model verification is an integral process of MDE concerned with checking models for known problems. Automating model verification is important as the process is resource intensive and error prone. This paper presents an approach (called pQVT) to automate the detection of problems in MOF-based models. pQVT specifies problems with a QVTr transformation from models conforming to a MOF-based DSML, where elements playing roles in problems are identified, to result models where problem occurrences are reported. The approach is declarative, leverages a standard formalism, applies to any MOF-based DSML, has well-defined detection semantics, has powerful reuse and modularization semantics and produces concise and detailed results.

In addition, the paper presents a catalog of 113 problems for MOF 2.4-based metamodels split into four categories: UML well-formedness, MOF well-formedness, semantic and convention. The catalog was formally specified using pQVT, which was found to be both adequate and concise. It was then used to detect problem occurrences in recent revisions of the standard UML metamodel. A large number of occurrences were detected and analyzed. Results show that pQVT is effective at finding real problems in realistic models as it led to a 53% reduction of the problem occurrences detected in the UML 2.4 metamodel, which were all verified and agreed upon by the UML RTF (a large majority of the identified problems resulted in changes, as we explained earlier). Finally, the case study shows that pQVT has a good performance as it could execute the entire catalog of problems on the complex UML metamodel in about one minute clearly demonstrating this is a scalable, practical technology.

Acknowledgements

The authors would like to thank the following individuals for helping review the case study: Steve Cook (Microsoft), Nicolas Rouquette (JPL) and Pete Rivett (Adaptive).

References

1. Unified Modeling Language (UML), Infrastructure v2.4. OMG ptc/2010-11-03.
2. Business Process Modeling and Notation (BPMN) v2.0. OMG dtc/2010-06-05
3. Meta Object Facility (MOF) Core v2.4. OMG ptc/2010-12-08.
4. Object Constraint Language (OCL) v2.2. OMG formal/2010-02-01.
5. Elaasar, M., Briand, L., Labiche, L.: An Approach to Detecting Design Patterns in MOF-Based Domain-Specific Models with QVT. Technical Report SCE-10-02, Carleton University, November (2010). (Submitted for publication)
6. Query/View/Transformation (QVT) v1.0. OMG formal/2008-04-03.
7. OMG issues database. http://www.omg.org/issues/
8. Fowler, M.: Refactoring: Improving the Design of Existing Code. 1st edition, June (1999).
9. Koenig, A.: Patterns and Antipatterns. J. of OO Programming 8(1), pp. 46–48 (1995).
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. (1995).
11. Brown, W., Malveau, R., Brown, W., McCormick III, H., Mowbray, T.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. 1st edition, (1998).
12. Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille J.: Consistency Problems in UML-Based Software Development. LNCS, vol. 3297, pp. 1-12, (2005).
13. Koehler, J., Vanhatalo, J.: Process Anti-patterns: How to Avoid the Common Traps of Business Process Modeling. IBM WebSphere Developer Technical Journal, Feb., (2007).
14. Travassos, G., Shull, F., Fredericks, M., Basili, V.: Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality. In Proc. of OOPSLA ‘99, pp. 47-56, (1999).
15. Dhambri, K., Sahraoui, H., Poulin, P.: Visual Detection of Design Anomalies. In Proc. of CSMR ‘08, pp. 279-283, (2008).
16. Marinescu, R.: Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proc. of the ICSM ‘04, pp. 350-359, (2004).
17. Munro, M.: Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In Proc. of the 11th Int'l Soft. Metrics Symposium, p.15, (2005).
18. Trcka, N., Aalst, W., and Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In Proc. of CAISE ‘09, LNCS 5565, pp.425 (2009).
19. Correa, A., Werner, C., Zaverucha, G.: Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns. LNCS, vol. 1844, pp. 33-191, (2000).
20. Moha, N., Gueheneuc, Y.-G., Duchien, L., Le Meur, A.-F.: DECOR: A Method for the Specification and Detection of Code and Design Smells. TSE, vol.36, no.1, Jan/Feb (2010).
21. Khomh, F., Vaucher, S., Gueheneuc, Y.-G., Sahraoui, H.: A Bayesian Approach for the Detection of Code and Design Smells. In Proc. of ICSQ ‘09, pp. 305-314, (2009).
22. Meyer, M.: Pattern-based Reengineering of Software Systems. In Proc. of WCRE ‘06, pp. 305-306, Oct. (2006).
23. Feng, T., Zhang, J., Wang, H., Wang, X.: Software Design Improvement through Anti-patterns Identification. In Proc. of ICSM ‘04, pp. 534, (2004).
24. Enckevort, T.: 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, (2009).
25. Medini QVT: A Toolset for Model to Model Transformations. http://projects.ikv.de/qvt
26. Elaasar, M., Briand, L. and Labiche Y.: "Metamodeling Anti-Patterns", 2010. https://sites.google.com/site/metamodelingantipatterns
27. UML Revision Task Force Wiki. http://www.omgwiki.org/uml2-rtf