An Approach to Detecting Design Patterns in MOF-Based Domain-Specific Models with QVT

MAGED ELAASAR¹, LIONEL C. BRIAND², AND YVAN LABICHE³

¹IBM Canada Ltd, Rational Software, Ottawa Lab
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com

²Simula Research Laboratory, Department of Software Engineering
Martin Linges v 17, Fornebu, P.O.Box 134, 1325 Lysaker, Norway
briand@simula.no

³Carleton University, Department of Systems and Computer Engineering
1125 Colonel By Drive, Ottawa, ON. K1S 5B6, Canada
labiche@sce.carleton.ca

ABSTRACT. A design pattern is a recurring and well-understood design fragment. In the context of a domain-specific modeling language (DSML), a design pattern is represented as a structure of constrained and inter-related model elements. Techniques that analyze models by detecting occurrences of known design patterns can simplify model comprehension and maintenance. Though each DSML may have its own unique set of design patterns, it is not practical to learn a separate detection technology for each specific DSML or family of design patterns. This paper describes a generic approach to specify domain-specific design patterns for MOF-based DSMLs at the metamodel level and automatically detect their occurrences in models. The approach is based on QVT (Query/View/Transformation). Patterns are specified declaratively with QVT-Relations (QVTr) transformations from DSML models, where elements playing pattern roles are identified, to a newly-defined DSML model for reporting identified occurrences. Pattern detection is implemented by executing these transformations. The approach has been prototyped using Eclipse technologies and used in a case study to detect the well-known GoF patterns in a design model of a large open-source system. Results were then analyzed for accuracy using precision and recall as metrics, confirming the adequacy of the approach to detect pattern occurrences with high accuracy.

KEYWORDS: Domain-Specific Model, Design Pattern, Specification, Detection, GoF, UML, QVT.

1 Introduction

Domain-specific modeling languages (DSML) [1] are high-level languages specific to a particular application domain. Unlike general-purpose modeling languages, such as UML [2], a DSML uses familiar terms and concepts to a domain allowing its users to learn it faster. Historically, the adoption of DSMLs had been impeded by the onerous task of building specific tools for them like (textual/graphical) editors, compilers, debuggers, translators, and analyzers. However, over the past decade, strong interest in model-driven engineering (MDE) has led research into developing technologies and tools that generically support the definition and use of DSMLs [3].

The main foundation upon which these technologies are built is the abstract syntax (AS) of a DSML. The abstract syntax describes language primitives whose semantics are familiar to all practitioners in a domain. One of
the most common formalisms for defining abstract syntaxes is the Meta Object Facility (MOF) [4]. A model
describing the AS of a DSML in MOF is called a metamodel. Examples of metamodel-based DSMLs include
BPMN 2.0 (Business Process Modeling Notation) [5] and Diagram Definition [6]. Even though UML is most
widely used for object-oriented software design, and thus can be considered domain-specific, it is not generally
classified as a DSML. On the other hand, the AS of UML is defined with a MOF metamodel too. Hence,
technologies and tools that apply generically at the metamodel level to MOF-based DSMLs equally apply to
UML and vice versa. In the context of this paper, we present an approach that falls into this category and thus
when we talk about technologies for MOF-based DSMLs we mean to include UML as well.

Another mechanism commonly used to define the AS of a DSML is a UML profile [7]. A profile allows UML
to be customized for a particular domain using a set of stereotypes (defining new attributes and constraints).
Examples of profile-based DSMLs include SysML [8], for systems modeling, and MARTE [9], for modeling real-
time embedded software. Unlike a metamodel, the syntax of a profile is defined with UML (vs. MOF).

One of the foundational technologies that work on the abstract syntax of a language is model analysis through
the detection of occurrences of known design patterns, which is the topic of this paper. A design pattern is a best
practice solution to a standard design problem. The notion was first introduced to software engineering by
Gamma et al. [10]. As a modeling and design tool, a design pattern specifies a structure of constrained and inter-
related model elements. In the context of a DSML, such structures need to be formally specified, indicating how
(DSML) model elements play roles in a solution to a domain-specific design problem. Each DSML may have its
own unique set of design problems and best practices [11]. For example, a DSML for modeling electronic circuits
may have a design pattern consisting of logic gates arranged and interconnected in specific ways to optimize a
particular function. Hence, it is important to have a technology that allows each DSML to specify and detect its
own unique design patterns. It is also desirable that such technology work consistently for all DSMLs to enable
the development of a generic tool, reducing the cost for tool developers and the learning curve for practitioners.
Conversely, some DSMLs may have a similar set of design problems and best practices and therefore it would
also desirable to specify their design patterns in a generic way that can be reused across those DSMLs.

An important motivation for the specification and detection of DSML design patterns is assisting users with
the comprehension and maintenance of DSML models. Modern systems are becoming larger, more complex and
often developed with tight deadlines and frequently changing requirements [12]. This means that design
documentation is often missing or, if available, may not exactly reflect the built system. The benefits of using
design patterns, as well-understood building blocks of design, would be compromised without knowledge of
where they have been used. Therefore, a technology that helps detect occurrences of design patterns in existing
models becomes important for the ongoing comprehension, maintenance and documentation of the system. Other
uses of this technology include detecting anti-patterns (design flaws) [13], checking design conformance [14] and
assisting pattern-based evolution [15].
Many approaches have been proposed to detect design patterns ([17] to [41]). Most existing approaches focus on detecting them in either the source code or a design model in UML. Some of these approaches express patterns at the model level (i.e. using an instance of the DSML) vs. the more generic metamodel (language) level. Other approaches need to be customized every time for new patterns. Such approaches cannot be reused for different patterns without a significant effort. Another important concern is the ability to manage the complexity of patterns. Different patterns often share common parts while others have several variants. Some approaches do not have reuse semantics (like composition or inheritance) and therefore are harder to use to support these cases. Another important aspect is familiarity and ease of implementation. Some approaches use formalisms that practitioners in the domain are not trained in (e.g., higher level logic calculus or neural networks). Others require technologies and/or tools (e.g., theorem provers) that are harder for tool developers to build or integrate with existing modeling infrastructure.

In this paper, we propose a generic approach to specify arbitrary design patterns for any DSML and detect their occurrences in models conforming to that DSML. We focus on design patterns of MOF-based DSMLs (those defined with a MOF metamodel). Profile-based DSMLs, being defined with a UML profile instead, are out of scope of this paper and outlined as future work in Section 6.3. We also focus on the ability to specify unique patterns for any DSML accurately. However, defining the same pattern generically on a number of DSMLs is an advanced topic that is out of scope of this paper and is outlined as a future work in Section 6.1.

Our newly proposed approach is based on model transformation techniques. More specifically, our approach (named pQVT) uses a QVTr (Query/View/Transformation Relations) [16] transformation where each pattern is specified declaratively as a relation between metaclasses of a given DSML playing roles in the pattern and metaclasses of a new DSML (named pResults) that we designed specifically for compactly representing pattern occurrences. Detection is then performed by executing the transformation on an input model, looking for occurrences of specified patterns, and generating a pResults model containing the detected occurrences.

Our approach offers several benefits that together make it stand out as a practical solution to the problem stated above. Firstly, QVTr is a transformation formalism that generically works on the abstract syntax of MOF-based DSMLs. This gives the approach the ability to specify patterns for arbitrary DSMLs without the need to customize the formalism or the detection algorithm. Secondly, QVTr is a highly declarative rule-based language. Every rule specifies a constrained relation between specific metaclasses in a DSML. This allows patterns, defined as rules, to concisely specify how pattern participants (instances of metaclasses) relate to each other and to the roles they play in pattern occurrences (instances of metaclasses in pResults). Thirdly, expressions in QVTr are specified using the Object Constraint Language (OCL), which is based on first-order predicate logic with set semantics and is often used as a constraint language when defining MOF-based DSMLs. Modeling practitioners who would be responsible for specifying patterns using our approach would typically be experienced individuals with knowledge of their domain and the way their DSML is expressed in MOF and therefore would also likely
know OCL. Fourthly, QVTr has built-in reuse semantics allowing rule composition and therefore pattern composition. QVTr also has support for utility queries which, along with rule composition allow for a modular approach to defining patterns where complex patterns are defined by composing simpler ones and where pattern variability is managed by refactoring out the common logic. Fifthly, since pQVT is based on QVTr, it has well defined execution semantics, which we reuse as pattern detection semantics (i.e., in our approach we do not define new detection algorithms; instead the QVTr transformations are the detection algorithms). Sixthly, QVTr has both a textual and a graphical concrete syntax. This allows pattern definitions to be specified/viewed using either notation depending on user preference and/or training. Lastly, QVTr is a standard language (defined by OMG) increasing the possibility for wide adoption and tool interoperability.

We prototyped pQVT using open-source Eclipse-based technologies that allowed for defining pResults and specifying/executing QVTr transformations. We also validated pQVT in a case study where we have specified a representative subset of the GoF patterns (typically used for object-oriented software design) on UML. We chose GoF because a lot of modeling practitioners are acquainted with them, to ease the comparison of our approach to others in the literature and because they have a level of complexity (number of roles, their inter-relationships and constraints) and variability (number of variants) that is likely to be found in domain-specific patterns in general. We also structured a set of experiments where those patterns were detected in UML models with varying levels of details (e.g. structural diagrams, behavioral diagrams and operations’ post conditions) to investigate which details are most critical to accurately detect GoF patterns. The models represented the design of a large open-source project. In each experiment, we assessed the accuracy of the detected pattern occurrences. Two standard metrics were used: precision—measuring what fraction of detected occurrences is real—and recall—measuring what fraction of real occurrences is detected.

Results of the case study suggest that pQVT can adequately specify GoF patterns with their various (structural and behavioral) constraints and multiple variants. They also suggest that GoF patterns can be most precisely detected in UML design models when considering both structural diagrams (e.g. class diagrams) and behavioral diagrams (e.g., sequence diagrams). While (OCL) post-conditions are useful for formalizing the behavior of operations, relying on them alone for behavioral checking (in the context of pattern detection) badly hurts recall, while checking them along with behavioral diagrams does not result in significant added benefits to accuracy. Finally, recall can be improved further when other pattern variants, that have been used in past modeling projects, are also specified and used for detection in future ones.

The rest of this paper is structured as follows: Section 2 highlights related work; QVTr is overviewed in Section 3; Section 4 describes the pQVT approach and demonstrates its applicability to various pattern situations; a case study featuring GoF patterns in UML is discussed in Section 5; Section 6 highlights the known limitations of the approach and discusses future work; Finally the contribution of this paper are summarized in Section 7.
2 Related Work

The research area of pattern specification and detection has been active for a number of years and resulted into a whole range of approaches. These approaches can be categorized along several axes: a) proposing new formalisms (steeper leaning curve and expensive to support) vs. adapting existing ones (building on available knowledge and tools) for pattern specification; b) using (easier to read) graphical vs. (easier to manipulate) textual vs. (having both advantages) hybrid notations to specify patterns; c) specifying patterns at the (easier for end-users to work with) model level vs. at the (easier to build generic tools for) metamodel level; and d) being suitable for specific patterns and/or DSMLs (requiring significant effort to reuse) vs. being applicable to arbitrary patterns and/or DSMLs (readily reusable). Pettersson et al. performed a detailed survey of approaches in this area and proposed a framework for assessing their accuracy [17]. Others surveys can be found in [18] and [19]. The rest of this section highlights some notable approaches in the literature and compares them to pQVT.

Some approaches convert data (e.g. source code) into representations that are amenable to analysis and then specify design patterns with predicate logic. For example, Seemann et al. [20] convert code into a graph representation, Huang et al. [21] represent data into a Prolog representation and Beyer et al. [22] represent data as relations encoded as binary decision diagrams. Birkner [23] converts data into facts, uses a SQL-like query language to define various relations between artifacts and then uses their combinations to match patterns. One issue with these approaches is the non-trivial process of defining data converters (deciding which information to extract and how to represent it in the new format), which mainly depends on the semantics of the data. In the context of DSMLs, since every DSML can have its own unique semantics, different converters would be required. Additionally, the need for this conversion process makes integrated tool support much harder to create. In contrast, pQVT allows patterns to still be specified in predicate logic (using OCL) directly on MOF-based metamodels and detected directly in models conforming to that metamodel.

Some approaches use other kinds of logic calculus. Mikkonen [24] uses the temporal logic of actions to formalize the temporal behavior of software. Taibi et al. [25] formalize structural features of code as relations between program elements, specify post-conditions with predicate Logic and describe operation behavior with temporal logic. The use of temporal logic is powerful for expressing behavioral aspects of software patterns but would need to be mapped into the semantics of every DSML, which is not typically a simple task for pattern authors. Eden’s [26] approach, called LePUS, defines an ontology of OO building blocks used to represent software data such as: entities (e.g., classes, attributes), collections (entities with multiplicities) and relations (e.g. inherit, invoke, create). The semantics of those blocks are formalized with higher order monadic logic. Patterns depicted by LePUS diagrams are formulae expressed in that logic that get solved to detect pattern occurrences. The LePUS blocks, however, are specific to OO systems and they cover only structural features.
Other approaches [27], [28] and [29], incorporate quantitative methods to partially optimize pattern detection by speeding up the matching of a subset of constraints. The idea is to devise a formula that assigns scores to model elements and pattern roles based on those constraints. Then by comparing scores of elements to those of roles, a candidate set of occurrences can be quickly detected. For example, the scoring formula devised by Dong et al. [27] takes advantage of a property of prime numbers that the product of prime numbers raised to powers is always a unique composite number. They use this property to assign a prime number to each structural element (e.g., attribute, operation) in a class and raise it to a power that equals the total number of such elements. The resulting product then represents a unique weight for an element (in this case a class). Relations between pairs of classes are encoded similarly using an \( n \times n \) matrix. Checking constraints (e.g., the minimum number of attributes in a class is 2) is then reduced to simple arithmetic operations over the weights and the matrix (e.g., find all classes with a specific weight or an integral multiple of it). This technique is used to optimize structural checking by finding an initial set of candidates and is followed by behavioral and semantic checking through code inspection of those candidates. The work was later extended in [30] to visualize instances of design pattern by annotating UML models directly with stereotypes showing role bindings. One obvious drawback of this optimization technique is its scalability. There is a physical limit to the number of constraints that one can encode. In addition, these encodings need to be done upfront on the whole model, i.e., the search for a pattern is unguided by the specific pattern requirements, which could be expensive. In contrast, patterns specifications in pQVT can have any number of constraints, allow efficient navigation of the models according to those constraints and do not require a big memory footprint (to hold the weights and matrix).

Graph transformation techniques also have been used to specify and detect design patterns. For example, Eppestein [31] specifies a design pattern as a graph whose nodes represent entities and whose edges represent relationships among entities. The detection is then done by graph isomorphism: identifying subgraphs similar to a template graph in a big graph, which is a difficult problem. Pettersson and Lowe [32] proposed transforming graphs of systems into planar graphs to improve performance with interesting results.

Furthermore, some approaches use modeling techniques. Le Guennec et al. [33] extend the UML 1.4 metamodel to incorporate collaboration occurrences. A pattern is defined as a UML collaboration constrained with OCL. Mak et al. [34] also define a pattern with collaborations modeled in UML after extending it with action semantics. Unlike pQVT, which defines patterns at the metamodel level, these approaches define patterns at the model (instance) level and thus are mainly applicable to a specific language (UML in this case).

Other approaches use metamodeling techniques. DPML [35] is a stand-alone language for modeling design pattern specifications and occurrences that defines its own metamodel and notation. Using DPML, pattern participants, their multiplicities and their constraints are specified. Constraints are either simple (specified informally in natural language) or binary directed relations (e.g., implements, extends). When relations are specified between participants with multiplicities, they are further marked as total, regular, complete or
incomplete based on the number of participants on each side that must be related to form a pattern. DPML focuses only on specifying structural constraints, has no reuse mechanisms and does not provide detection semantics.

RBML [36] is another metamodeling approach that defines UML patterns as a specialization of the UML metamodel. Pattern roles are metaclasses specializing ones from UML and have their well-formedness rules expressed in OCL. Structural aspects of patterns are represented as a specialization of class diagrams, while behavioral aspects are represented as a specialization of behavioral diagrams. The approach enjoys some benefits similar to pQVT in terms of specifying patterns at the metamodel level and using OCL for constraints. However, different diagrams (e.g. class, sequence and state-charts) have their own specific detection semantics. This means that when generalizing the approach to other DSMLs, specific detection semantics need to be devised for each DSML, which is not trivial. On the other hand, pQVT does not define separate detection semantics as it uses the execution semantics of QVTr generically for detecting pattern specified for any DSML.

Elaasar et al. [37] propose the Epattern language, which extends MOF with pattern specification semantics and propose a visual notation for that specification. Similar to pQVT, the approach can specify patterns of any MOF-based DSML. However, only simple patterns are demonstrated and no detection semantics are provided.

Beyley and Zhu [38] define a new metamodeling language based on first order predicate logic with a notation based on a graphical extension of BNF. DSMLs are expressed in this language and patterns are expressed using formulae in the same logic formalism. The approach has been used to define all GoF patterns with their structure and behavior. The approach also has facilities to express variants without having to define new patterns. In comparison, pQVT uses the same expressive power (i.e., first order predicate logic, thanks to OCL/QVT), albeit it works directly on models in their standard MOF-based representation.

Gueheneuc and Antoniol [39] propose DeMIMA, a three-layered architecture to detecting patterns from source code, where the first and second layers are geared towards accurately reverse engineering the code into a model conforming to a new metamodel they defined. In the third layer, design patterns are expressed using the same metamodel. Pattern definitions are then transformed into a constraints system consisting of variables (pattern roles), domains of variables (types of elements playing those roles) and constraints (relationships among the roles). The constraint system is resolved to match patterns. An interesting feature of this approach is that it can detect partial occurrences of patterns where some constraints are not satisfied. This allows pattern authors to interactively refine specifications by relaxing constraints. In comparison, pQVT also allows specifications to be defined using variables, domains and constraints but does so directly in QVTr without introducing a new language. The execution semantics of QVTr plays the same role as the constraint resolution semantics of DeMIMA. In addition, pQVT works on data expressed directly in their own metamodel. No conversion is needed.

Briand et al. [40] proposed a technique, while not used to detect design patterns per se, is used to detect places in models where patterns could have been applied. The work was demonstrated on GoF patterns in UML. The idea is to specify decision trees whose nodes represent checks that can be automated (with OCL) or used to collect
feedback from a user. The leaf nodes then provide suggestions for design patterns to apply. Although the technique could also be used to detect patterns, writing decision trees is considered imperative. pQVT on the other hand is declarative approach, making patterns much easier to specify.

Transformation-based approaches also exist. MTF [41] is a model-based transformation language whose roots can be traced back to QVTr. Similar to pQVT, MTF is used to specify patterns as relations between metaclasses of MOF-based DSMLs. Each relation declares one or more parameters typed by metaclasses that play roles in the pattern and can be constrained. MTF also provides relation reuse semantics such as composition and inheritance. However, MTF’s query language is very basic and limited to simple navigations over model relations (e.g., `c.attribute1.attribute2`) and few predefined constraints (e.g., `Equals`, `MatchString` and `InstanceOf`). Any other needed expression or constraint needs to be defined by language extensions (implemented in Java). In contrast, pQVT uses the full expressive power of OCL/QVTr. Also, unlike pQVT, MTF is unable to express local variables in relations, which leads to MTF’s transformations being more verbose than pQVT’s.

As can be seen from the review above, existing approaches vary in their ability to adequately address the problem of generic pattern specification and detection for DSMLs. For example, some require models to be transformed into other representations making it less scalable; others depend on formalisms (e.g., high order logic) that average modelers are not trained in; others express patterns at the model level making it specific to one DSML or require custom detection algorithms for each DSML; and others lack complexity management semantics needed to produce concise and modular specifications. On the other hand, we believe that our pQVT approach has a combination of capabilities that together make it a more adequate solution to the problem: it reuses an existing standard formalism (i.e. QVTr) lowering the learning curve and enhancing tool interoperability; the formalism is declarative making the specifications very concise; it has a hybrid textual/graphical notation making specifications easy to manipulate and understand; it specifies patterns at the metamodel level allowing for building generic tools; and it works on models directly in their original MOF-based representation.

3 Overview of QVTr

The Query/View/Transformation (QVT) [16] is a specification by the Object Management Group (OMG). It defines a standard way to transform between MOF-based models using model transformation rules. QVT defines three DSMLs: Core, Relations and Operational. The last two are declarative languages defined at different levels of abstraction (Relations being defined at a higher level). The Operational language is an imperative language that provides constructs commonly found in imperative languages, such as loops and conditions. In our approach, we chose to use QVT-Relations (QVTr) for pattern specification due to its declarative syntax and higher level semantics. The abstract syntax of QVTr is defined using minimal extensions to OCL and MOF. Moreover, the language has both a textual and a graphical concrete syntax. All of this is expected to simplify pattern specifications as demonstrated in Section 4. The remainder of this section overviews QVTr providing the level of
details that we feel will be necessary in the remainder of the paper. We will use a running example all through the section to illustrate the concepts: the transformation of a UML class diagram into a database schema.

### 3.1 Transformation and Model Types

In QVTr, a transformation between two candidate models is specified as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a MOF-based DSML. Candidate models are declared with names and types, representing the DSMLs that restrict the kinds of elements they can contain. An example in Figure 1 declares a transformation named `UMLToRDBMS` between two candidate models: `uml`, declaring UML as its DSML (type), and `rdbms`, declaring RDBMS (a language for database schema definition) as its DSML. The transformation contains a number of relations (rules) specifying its logic.

```plaintext
transformation UMLToRDBMS {uml : UML, rdbms : RDBMS} {
    relation PackageToSchema {...}
    relation ClassToTable {...}
    relation AttributeToColumn {...}
}
```

Figure 1 – The Transformation UMLToRDBMS Summarized

A transformation can be invoked either to check two models for consistency (based on the relations) or to modify one model to enforce consistency. A transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as the target. The execution attempts to make the relations hold by creating, deleting, or modifying elements in the target model only.

### 3.2 Relations and Domains

Relations in a QVTr transformation declare constraints that must be satisfied by elements of the candidate models. A relation is specified with a name, an ordered set of parameters (called domains) that represent related elements from the candidate models, and a set of constraints on those parameters. The reason a parameter is called a domain refers to basic definitions in mathematical functions. Assume you have a real function \( f(x) \). The domain of \( f(x) \) is the set of legal values of \( x \) that make the function output real values. In other words, the domain of a function is the set of constraints on the values of its parameters. Similarly, in QVTr, a parameter of a relation, through its constraints, specifies the set of model elements that can bind to it, and thus is called a domain of the relation. QVTr relations can have more than one domain since they can be executed in more than one direction. (In contrast, in mathematics, a function has one domain and one range since it executes always in one direction.) The constraints that can be specified for a domain include its type (a metaclass that is the type of elements bound to that domain), specific values for its attributes and specific association links to other related elements.

Relation domains also represent the set of elements that must be located, modified, created or deleted in a candidate model in order to satisfy the relation. An example is the relation `PackageToSchema`, in Figure 2, between a UML package and an RDBMS schema because we want to transform packages into schemas. The relation is declared with two domains that match elements in the `uml` and `rdbms` models, respectively. Each
domain specifies a simple element: a UML package \( p \) and a RDBMS schema \( s \). Both elements are specified with their \textit{name} attributes bound to the same variable \( \text{pn} \) implying that they should have the same value.

```
relation PackageToSchema { /* maps each package to a schema */
    domain uml p:Package { name=\text{pn};
    domain rdbms s:Schema { name=\text{pn} };}
}
```

\textbf{Figure 2 – The Relation PackageToSchema}

A relation can also define nested variables within domains representing model elements constrained to be involved in relations with the domain elements through specific association. Unlike domains, nested variables are not considered as parameters of a relation, but more like local variables. For example, Figure 3 shows a relation ClassToTable between a domain \( c \) of type \textit{Class} and a domain \( t \) of type \textit{Table}. Notice that the domain \( c \) is nesting a variable \( p \) of type \textit{Package} and is related to it through the package association (from the UML metamodel). Similarly, the domain \( t \) is nesting variable \( s \) of type \textit{Schema} and is related to it through the schema association (from the RDBMS metamodel). While these examples show direct nesting by a domain, variables can also be nested recursively to any depth. Variables are uniquely identified by their names in the scope of a relation. This means that all occurrences of a variable with the same name in a relation must refer to the same value. Repeating variables in more than one place in the relation (e.g., the variable \( \text{pn} \) in Figure 2, variable \( \text{cn} \) in Figure 3) is often used to establish traceability between different elements in a relation.

A relation can be constrained by two sets of predicates, a \textit{when} clause and a \textit{where} clause. The \textit{when} clause specifies the conditions under which the relation needs to hold. For example, the relation ClassToTable (Figure 3) needs to hold between class \( c \) and table \( t \) only when the PackageToSchema relation holds between package \( p \) (containing \( c \)) and schema \( s \) (containing \( t \)). The \textit{where} clause is used to extend a relation to specify other relations that must hold when this relation holds. For example, whenever the relation ClassToTable holds between class \( c \) and table \( t \), the relation AttributeToColumn needs also to hold between them. This effectively makes the \textit{when} and \textit{where} clauses two forms of relation composition, where the domains of the composed relation get bound to values from the composing relation. As mentioned earlier, the domains of a relation is an ordered set, hence when composing a relation, the values need to be bound to the domains in the same order. For example notice how the composition of relation PackageToSchema in the \textit{when} clause (Figure 3) binds variables \( p \) and \( s \), respectively, to the domains of the relation in Figure 2.

```
relation ClassToTable { /* maps each concrete class to a table */
    domain uml c:Class {
        name= \text{cn},
        isAbstract= false,
        package = p:Package{};
    };
    domain rdbms t:Table {
        name= \text{cn},
        schema = s:Schema{};
    };
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t); }
}
```

\textbf{Figure 3 – The Relation ClassToTable}
A transformation contains two kinds of relations: top-level (marked as top) or non-top-level. The execution of a transformation requires that all top relations hold, whereas non-top relations are required to hold only when they are composed directly or transitively from the where clause of another relation. In our example (Figure 4), PackageToSchema and ClassToTable are top relations, whereas AttributeToColumn is a non-top relation (composed by the where clause of ClassToTable).

```
transformation uml2rdbms (uml : UML, rdbms : RDBMS) {...
top relation PackageToSchema {...}
top relation ClassToTable {...}
relation AttributeToColumn {...}
...}
```

Figure 4 – The Transformation UMLToRDBMS with Top and Non-Top Relations

As mentioned in Section 3.1, a transformation can be executed to enforce consistency between candidate models by choosing one of them as target. Whether each relation is going to be enforced is determined by its target domain being marked as checkonly or enforce. When it is marked as checkonly, the execution just checks to see if there exists a valid match in the relevant model that satisfies the relationship. However, when it is marked as enforce and if the check fails, the target model is modified so as to satisfy the relationship. In the example relation (Figure 5), the domain for the uml model is marked as checkonly and the domain for the rdbms model is marked as enforce. This means that assuming we are executing the relation from the rdbms domain to the uml domain, and there exists a schema in rdbms for which we do not have a package in the uml model with the same name, then this is reported as an inconsistency between the two domains. However, assuming we are executing the relation in the direction of the rdbms domain, and there exists a package in the uml model for which there is no schema in rdbms model with the same name, a new schema is created and given that name.

```
relation PackageToSchema { /* maps each package to a schema */
    checkonly domain uml p:Package { name=pn }
    enforce domain rdbms s:Schema { name=pn }
}
```

Figure 5 – The Relation PackageToSchema with CheckOnly and Enforce Domains

### 3.3 Graphical Notation

QVTr allows relations to be specified using a graphical notation that is based on the UML object diagram notation. This is due to the relation’s domains representing a collection of objects, links and values, the type of information typically visualized by UML object diagrams. The example in Figure 6 shows the ClassToTable relation, defined previously (Figure 3), using a graphical notation. The relation is represented using a rectangular shape with a hexagon shape in the middle representing the two related candidate models uml and rdbms. The C annotation means that the uml domain is marked checkonly, while the E annotation means the rdbms domain is marked enforce. Elements in both domains are represented by object shapes on both sides of the hexagon with the domain objects marked with keyword «domain». Attribute values are specified in separate compartments in the object shapes and association links are represented as lines connecting object shapes. The where and when clauses have their own compartments within the main relation shape.
3.4 Object Creation and Keys

As mentioned in Section 3.2, a domain marked as enforce serves to locate, update, delete or create objects in a target model. However, when creating objects we want to ensure that duplicate objects are not created, for instance by different relations, when the required objects already exist. While MOF allows a single attribute of a class to be nominated as a unique identifier of instances, in most metamodels this is not sufficient. That is why QVTr introduces the concept of a key, which defines a set of attributes of a class that uniquely identify an object instance of the class in a model. A class may also have multiple keys (as in relational databases). In our running example, we can specify that in a RDBMS model, a Schema is uniquely identified by its name attribute, while a Table is uniquely identified by two attributes: its name and the schema it belongs to, as shown below.

![Figure 7 – The Unique Keys of RDBMS](image)

3.5 Transformation Queries

Different QVTr relations often specify similar if not identical when and/or where constraints. Since repeated code is hard to maintain and tend to increase the size of transformations, QVTr introduces the concept of reusable queries. A query is a side-effect-free operation (with an ordered set of parameters) owned by a transformation and can be invoked (with an ordered set of values) from any OCL expression in the transformation, i.e., any when or where constraint. The body of a query can be specified by an OCL expression, or it may be ommitted to indicate it is implemented externally to the QVTr language in a platform-dependent way (the term for this in the QVT spec is a black-box query) like with Java. The example in Figure 8 shows an example query called areAllAttributesSingular, whose body is specified in OCL to check if a given UML class has all its attributes as singular, i.e., with a lower bound smaller than or equal to 1 (Figure 17 shows a subset of UML
metamodel where the relations used in this OCL expression are defined). The query is invoked from the when clause of relation ClassToTable, i.e., mapping only those classes with singular attributes to a table.

```oclmplate
relation ClassToTable { /* maps concrete classes with singular attributes only */
    domain uml c:Class {...};
    domain rdbms t:Table {...};
    when {
        areAllAttributesSingular(c);
        PackageToSchema(p, s);
    }
    where { AttributeToColumn(c, t); }
}
query areAllAttributesSingular (c : Class) : Boolean {
    c.ownedAttribute->forAll(a|a.lower <= 1)
}
```

Figure 8 – The Relation ClassToTable using a Query As a Condition

### 3.6 Execution Semantics

As mentioned in Section 3.1, QVTr transformations can be invoked to check or enforce consistency between candidate models. In both cases, the transformations have well-defined execution semantics. In this section, we summarize the enforcement semantics as they are used as detection semantics for pQVT (Section 4). Recall that a transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as a target. In this case, all top relations get executed. As top relations are not ordered, the only constraint on their execution order is that when relation A composes relation B in its when clause (top relations cannot be composed in where clauses) then relation A cannot finish executing until relation B finishes executing first.

When a relation is executed, the execution engine first tries to bind elements from the source model(s) to variables defined by the relation’s source domains. To do that, the engine searches the source models for a combination of elements that satisfy those variables’ constraints including those in the when clause. For each unique combination, the engine enforces the relation’s target domain (assuming that the domain is marked enforce). This results in possible changes to the target model to satisfy the constraints on the variables defined by the target domain including those constraints in the where clause. If executing a relation resulted into a binding of values to its domains, the relation is said to hold for these values. Furthermore, when a relation is composed in a when clause as a constraint, the engine decides on the constraint by checking whether the composed relation holds for the values bound in that composition. However, when a (non-top) relation is composed in a where clause, the engine executes such a relation by binding the values of the composition to the domains of the relation.

### 4 Details of pQVT

The main contribution of this paper is a new approach, called pQVT, which allows practitioners of any MOF-based DSML to specify design patterns specific to that DSML and detect them in their user models. The approach allows practitioners to specify design patterns with a QVTr transformation between metaclasses from a DSML, playing roles in patterns, and metaclasses from a newly-defined general-purpose pattern result DSML, called
pResults. The latter allows the organization of the detected pattern occurrences in a compact tree data structure to facilitate their inspection by users.

In this section, we show how pQVT leverages the concise syntax and rich semantics of QVTr to specify design patterns precisely and to detect most of their occurrences. We start by describing the new pResults DSML in Section 4.1. In Section 4.2, we introduce the Adapter design pattern as an example pattern (from the GoF family) for an example MOF-based language (UML in this case). To ease the understanding of the approach, we use the Adapter pattern, as a running example, to gradually show in Section 4.3 the thought process and techniques used to define an accurate (in terms of precision and recall) pattern specification.

4.1 pResults Language

In this section, we present pResults, a new DSML that we defined for compactly representing occurrences of any design pattern for any MOF-based DSML. A pattern occurrence is a unique mapping of pattern roles to the elements playing those roles in a model. A common representation for a pattern occurrence in the literature is a flat collection of role bindings [17], where each role binding specifies which model element plays which role. Assume we detect occurrences of the GoF Composite design pattern, which has three roles: Component (Role1), Composite (Role2) and Leaf (Role3). Suppose we find that class E1 is a Component, that it has two Composites (E2 and E3) and two Leaves (E4 and E5). Figure 9 (left) shows that four unique collections of role bindings would be identified, one for each triplet (Component, Composite, Leaf). The main disadvantage of this representation is scalability with potentially huge number of unique collections with similar role bindings. This can stretch the computing resources required for the detection process and can consume excessively long time from a user to inspect the results.

A better alternative, provided by pResults, is to represent pattern occurrences as a tree of role bindings: Figure 9 (right). In such a tree, each role is bound in a different level resulting in a unique occurrence being a complete branch from the root of the tree down to a leaf. This representation allows occurrences with similar role bindings to share sub-branches, thus reducing the overall footprint of the result. In addition, by ordering the levels of role bindings from highest to lowest (based on their importance) in the tree, a user can inspect the result by selectively drilling down the tree (gradually revealing finer details).

In order to represent this tree structure of pattern roles, we defined the MOF class diagram (metamodel) of pResults (Figure 10). The class Category represents a namespace for pattern occurrences. It has a name attribute and a pattern collection. The class Pattern designates a specific pattern in a category and has a name. For example, the Composite pattern would be represented as an instance of Pattern named “Composite” which is owned by an instance of Category named “Structural”. A pattern also has a rootBinding collection, representing the root nodes of each occurrence (tree of role bindings) of the pattern. The class RoleBinding has a role attribute representing the name of a role in the pattern and an element attribute referencing a model
element (in the DSML model) bound to that role. The latter is typed by MOF::Element, which is the implicit super class of all MOF-based DSML metaclasses, allowing referencing any domain-specific model element. The class RoleBinding also has a childBinding collection, representing role bindings at the next level down an occurrence (tree of role bindings), and a Boolean attribute isMain indicating whether the role is a main one. Although all role bindings in a pattern occurrence contribute to its uniqueness, the main role bindings are the subset that conveys the most important information about it.

![Diagram of role bindings and main role bindings](image)

Figure 9 – Pattern Occurrences as Collections (Left) or as a Tree (Right)

4.2 Example Pattern

In order to facilitate the description of pQVT, we use the Adapter pattern, one of GoF design patterns, as a running example. Like most GoF patterns, the Adapter pattern is often used in the context of object-oriented software design with UML. Although the pattern is classified as structural in the GoF catalog, it actually includes both structural and behavioral constraints making it representative of the complexity found in the catalog as a whole. In addition, while most GoF patterns are specified with one official variant in the original text, the Adapter pattern is specified with two variants, Class Adapter and Object Adapter, making it a more compelling example to study in the context of pattern variability.

The main motivation for using the Adapter pattern is to translate one interface for a class into another compatible interface. In this way, an adapter allows classes to work together that normally could not because of incompatible interfaces by translating calls to its interface into calls to the original interface. Figure 11 shows the two official variants of the Adapter pattern using UML class diagrams. In both variants, the new Interface is
represented by the abstract class Target defining the abstract operation request. Similarly, the original interface is represented by the class Adaptee defining an operation realRequest. In the Class Adapter variant, the Adapter class specializes both classes and redefines the request operation to delegate to the inherited realRequest operation. In the Object Adapter variant, the Adapter class specializes the class Target, aggregates the class Adaptee, and redefines the operation request to delegate to the realRequest operation on the aggregated instance anAdaptee.

Figure 11 – Two Variants of Adapter Pattern: Class Adapter (left) and Object Adapter (right)

4.3 Pattern Specification

In this section, we gradually show the thought process and techniques of specifying design patterns with pQVT.

4.3.1 Getting Started

The first step is identifying the different roles in the design pattern and the types (metaclasses) of elements playing those roles. For example, based on the semantics of Adapter presented in Section 4.2, the pattern roles (common to both variants) are Target, Adapter and Adaptee (all of type UML::Class) and request and realRequest (both of type UML::Operation). At this point, the pattern can start to be specified as a relation in a QVTr transformation between a model conforming to the DSML of the pattern (UML in this example) and a model conforming to pResults. The relation specifies how classes of the DSML, playing roles in the pattern, map to a pattern occurrence (a tree of role bindings) represented in pResults.

For example, Figure 12 shows a QVTr transformation named GoF between a model named uml (conforming to UML) and a model named presults (conforming to pResults). The transformation defines a top relation named AdapterPattern to specify the Adapter pattern. Each role in the pattern is represented by a separate uml domain (in the relation) named after the role name and typed by the metaclass playing that role. In this case, five uml domains are defined: Target, Adapter and Adaptee (all typed by Class), and request and realRequest (both typed by Operation). The relation also defines a single presults domain that specifies a pattern occurrence using a tree of pResults variables. The tree starts at the top with variable c of type Category (variable c is chosen as a domain since it is the top most) with a name value of “Structural”. The category c nests a variable p of type Pattern with a name value of “Adapter”. The pattern p nests a tree of variables (rb1
to rb5) of type RoleBinding with name values equal to the role names (e.g., rb1’s name is “Target”) and element values equal to variables with identical names to those of the corresponding uml domains (e.g., rb1’s element is bound to Target). The RoleBinding variables are ordered down the tree in a way that reflects the relative importance of the role they represent. Notice that the uml domains of the relation are marked as checkonly, as the uml model is not expected to be modified during pattern identification, i.e., during execution of the transformation, while the presults domain is marked as enforce, as the presults model is expected to be modified (to report results). The relation AdapterPattern is also shown in graphical notation in Figure 13.

```
transformation GoF (uml : UML, pResults : pResults) {
  top relation AdapterPattern {
    checkonly domain uml Target:Class {};
    checkonly domain uml Adapter:Class {};
    checkonly domain uml Adaptee:Class {};
    checkonly domain uml request:Operation {};
    checkonly domain uml realRequest:Operation {};
    enforce domain pResults c:Category {name='Structural',
      pattern = p:Pattern (name='Adapter',
        rootBinding = rb1:RoleBinding {role='Target', element=Target:Class{},
        childBinding = rb2:RoleBinding {role='Adapter', element=Adapter:Class{},
        childBinding = rb3:RoleBinding {role='Adaptee', element=Adaptee:Class{},
        childBinding = rb4:RoleBinding {role='request', element=request:Operation{},
        childBinding = rb5:RoleBinding {role='realRequest', element=realRequest:Operation{}}}}})
  }
}
```

Figure 12 – GoF Transformation with Relation AdapterPattern

Figure 13 - Relation AdapterPattern in Graphical Notation

When the transformation is executed in the direction of presults, the AdapterPattern relation gets enforced (i.e., changes could be made to the target model to enforce consistency) because the presults domain (marked as enforce) becomes the target domain and the uml domains become the source domains. The execution first tries to find combinations of elements from the uml model that satisfy the constraints of the uml domains. The execution first tries to find combinations of elements from the uml model that satisfy the constraints of the uml domains. (Notice that so far the only constraints being defined in the relation are the types of the uml domains; more
constraints will be added in the next sections.) For each unique combination, the execution enforces the \texttt{presults} domain (i.e., locates, modifies, creates or deletes objects in the \texttt{presults} model) to satisfy the domain’s constraints. Before the transformation is executed for the first time, the \texttt{presults} model is empty and thus enforcing the \texttt{presults} domain leads to the creation of the tree of elements (i.e., the pattern occurrences) specified by the domain variables. The tree includes \texttt{RoleBinding} elements, whose \texttt{element} attributes are specified to equal variables with identical names to those of the \texttt{uml} domains. This results in those attributes being bound to the same combination of UML elements that was bound to the \texttt{uml} domains. Figure 14 shows an example of binding the variables of the \texttt{AdapterPattern} relation to a conforming combination of UML elements. On subsequent executions of the transformation, the \texttt{presults} model gets modified to report the detection results at the time. When no conforming combination of UML elements has been detected, pattern occurrences are not created (or get deleted if they were created previously).

![Figure 14 – A Unique Binding of AdapterPattern Objects to UML Elements](image)

In order to avoid creating duplicate elements in the \texttt{presults} model, we define unique keys for the \texttt{pResults} metaclasses, as shown in Figure 15. Notice that the keys include attributes that represent the possible containers of the classes (derived from the \texttt{pResults} metamodel in Figure 10). For example, the key for the class \texttt{Pattern} includes the container attribute \texttt{category}. Similarly, the two keys for class \texttt{RoleBinding} include the container attribute \texttt{pattern} in the first key and the container attribute \texttt{parentBinding} in the second key. The reason we included these container attributes is to ensure that the created objects are locally unique under their containers vs. globally unique across the whole model. Global uniqueness is not needed since, for example, the same role binding (i.e. role name and element reference) may be created indifferent role binding branches.

```
key \texttt{pResults::Category} { name };
key \texttt{pResults::Pattern} { name, category };
key \texttt{pResults::RoleBinding} { role, element, pattern };
key \texttt{pResults::RoleBinding} { role, element, parentBinding };
```

![Figure 15 – The Unique Keys of pResults](image)

4.3.2 Improving Precision

Having prepared the basic skeleton of a pattern specification in QVTr, the next step is to improve precision, i.e., the ratio of detected occurrences that are actually valid, by adding constraints to the pattern relation. So far, the \texttt{uml} domains of the relation have only been constrained by their types (i.e., the DSML metaclasses playing roles in the pattern). For example, the relation in Figure 12 binds any class from a UML model to the \texttt{Target},
Adapter and Adaptee roles and any operation to the request and realRequest roles. This is obviously not precise enough. To improve precision, we need to discover and add the missing constraints to the uml domains.

One way to discover the missing constraints is by analyzing the textual and/or graphical (e.g., a class diagram) description of the pattern. However, the description is often ambiguous, a problem we will address in Section 4.3.3. For now, we try to deduce basic constraints from the description to enhance the precision of the specification. For example, analyzing the description of the Adapter pattern in Section 4.2 indicates that there are two variants of the pattern that share some constraints and differ in others. The shared constraints include that the Target class is abstract and owns an abstract request operation, that the Adapter class has a generalization to the Target class and redefines the inherited request operation (by a concrete one) and that the Adaptee class owns a realRequest operation. The specific constraints for the Class Adapter variant include that the Adapter class has a generalization to the Adaptee class and that it implements the request operation to delegate the call locally to the realRequest operation inherited from Adaptee. The specific constraints for the Object Adapter variant include that the Adapter class aggregates the Adaptee class and that it implements the request operation to delegate the call to the realRequest operation on the aggregated anAdaptee object.

A convenient way to add these constraints is to leverage the relation composition mechanism of QVTr (Section 3.2). Specifically, we need to add the shared constraints to the AdapterPattern relation and define two new relations ClassAdapterPattern and ObjectAdapterPattern, which extend the former relation by being composed in its where clause, as shown in Figure 16. Each composition binds values from the composing relation to the domains of the composed relation. In this case, we bind the domains of AdapterPattern relation to the corresponding domains of the variant relations. This has the effect of further constraining the domains by the constraints specified in the variant relations. Furthermore, we need to refactor the AdapterPattern relation by moving the presults domain down to each of the variants relations. The reason for that is because we want the presults model to get modified only when we have considered all the constraints of each variant. Notice that in the composition of the two variant relations Figure 16 bottom), their presults domain is bound to the value null because no value exists for it in the AdapterPattern relation.

Adding the constraints to the domains require knowledge of the metamodel of the DSML. For example, Figure 17 shows a class diagram depicting a subset of the UML 2 metamodel that is relevant to the constraints of the Adapter pattern. The figure contains various attributes and associations that are used to constrain the pattern specifications. For example, the Target class in Figure 16 (top) is constrained to have a value of true for its Class::isAbstract attribute effectively requiring the class to be abstract. Another example is the Adapter class in the same figure that is constrained to own an operation request2 in its Class::ownedOperation collection. The operation is constrained to be concrete (i.e. BehavioralFeature::isAbstract equals false) and to reference the inherited operation request from its Operation::redefinedOperation collection.
The new variant relations (Figure 16 bottom) add constraints specific to their variants. For example, 
ClassAdapterPattern adds a constraint to class Adapter requiring it to have Adaptee as a superclass. 
Also, ObjectAdapterPattern adds a constraint to property anAdaptee requiring it to have class Adaptee as a type and to have its multiplicity (lower and upper) as 1. Notice that operation request2 is also defined as a domain in each of the variant relations so that it can further be constrained. For example, relation
ClassAdapterPattern defines a *when* clause constraint requiring operation request2 to call inherited operation realRequest directly. The constraint is defined by query directLocalCall (a local call is made to an operation on the same object). Similarly, relation ObjectAdapterPattern defines a *when* clause constraint requiring operation request2 to call operation realRequest on attribute anAdaptee directly. The constraint is defined by query directDelegationCall (a delegation call is made to an operation on an aggregate object).

Both directLocalCall and directDelegationCall are specified in more detail in Figure 18 (lines 1 and 2). Their logic is based on inspecting the behavioral information, if available in the model, to determine if one operation (caller) calls another operation (callee). We choose to inspect two kinds of behavioral information for operations: post-conditions (expressed in OCL) and behavior (expressed with sequence diagrams). (Notice that UML 2 calls the behavior of an operation its *method*.) If none of this information is available, the queries return true. We chose to do this knowing that it could harm precision (a problem we deal with in Section 4.3.4) but it would preserve recall in this particular case.

For post-conditions, we analyze the abstract syntax tree (AST) of the OCL expressions looking for calls to the callee operation. For example, consider operation Account::totalFee():Integer with a post-condition: result = variableFee() + plan.fixedFee(). Analyzing the AST for this expression (Figure 19), we find CallOperationExp nodes for operation variableFee on the source variable self (i.e., a local call) and operation fixedFee on the source property plan owned by self (i.e., a delegation call). However, QVTr does not directly support analyzing OCL expressions, thus we implement this analysis using a QVTr black-box query (lines 5 and 6 in Figure 18). Recall from Section 3.2 that such a query is defined with no body in the transformation and the actual body is provided externally in an implementation-dependent way. In this case, the implementation (not shown here) is done with the Java API of the MDT OCL project [42].

On the other hand, analyzing sequence diagrams can be specified using queries with OCL bodies. Figure 20 shows a sequence diagram specified as a behavior for the same operation Account::totalFee():Integer discussed earlier. Notice that specifying operations’ behavior with sequence diagrams is a design practice that is not necessarily adopted by mainstream textbooks on OO software engineering (e.g., [48] and [49]). Instead those books promote the description of sequence diagrams for use cases. In such sequence diagrams, however, we would still be able to find the specification of behavior in terms of messages being exchanged between classes. The OCL queries analyzing such sequence diagrams being cumbersome to show here, we chose to show a simpler, though equally feasible, alternative: the queries at lines 3 and 4 of Figure 18.

The logic of these queries is to take the Interaction (sequence diagram) representing the caller operation’s *method* (i.e., behavior) and search inside it for a *synchCall* message that references the callee operation as a *signature*. The message starts from a lifeline representing an instance self of the class. If the message (e.g., variableFee) ends on the same lifeline, then it represents a local call. However, if the message (e.g., fixedFee) ends on a lifeline representing the link element, then it represents a delegation call.
1. query directLocalCall(caller:Operation, callee:Operation) : Boolean {
   (caller.postcondition->size() = 0 and caller.method->isEmpty()) or
caller.postcondition->exists(c| directLocalCall(c, callee)) or
directLocalCall(caller.method.oclAsType(Interaction), callee)
}
2. query directDelegationCall(caller:Operation, link:TypedElement, callee:Operation) : Boolean {
   (caller.postcondition->size() = 0 and caller.method->isEmpty()) or
caller.postcondition->exists(c|  directDelegationCall(c, link, callee)) or
directDelegationCall(caller.method.oclAsType(Interaction), link, callee)
}
3. query directLocalCall(interaction:Interaction, callee:Operation) : Boolean {
   interaction.message->exists(m|
   m.messageSort = MessageSort::synchCall and m.signature = callee and
   m.sendEvent.oclAsType(MessageOccurrenceSpecification).covered->exists(l|l.name='self') and
   m.receiveEvent.oclAsKindOf(MessageOccurrenceSpecification) and
   m.receiveEvent.oclAsType(MessageOccurrenceSpecification).covered->exists(l|l.name='self'))
}
4. query directDelegationCall(interaction:Interaction, link:TypedElement, callee:Operation):Boolean {
   interaction.message->exists(m|
   m.messageSort = MessageSort::synchCall and m.signature = callee and
   m.sendEvent.oclAsType(MessageOccurrenceSpecification).covered->exists(l|l.name='self') and
   m.receiveEvent.oclAsKindOf(MessageOccurrenceSpecification) and
   m.receiveEvent.oclAsType(MessageOccurrenceSpecification).covered->exists(l|l.represents=link))
}
5. query directLocalCall(c:Constraint, callee:Operation):Boolean;/*black-box*/
6. query directDelegationCall(c:Constraint, link:TypedElement, callee:Operation):Boolean;/*black-box*/

Figure 18 – The Definition of the directLocalCall and directDelegationCall Queries

Figure 19 – The AST of the OCL Expression: result = variableFee() + plan.fixedFee()

Figure 20 – A Sequence Diagram Representing the Behavior of Operation totalFee
4.3.3 Improving Recall

In the previous section, we discussed how to improve the precision of a pattern specification by adding constraints to the QVTr relation. Constraints ensure that only valid combinations of elements are bound to the roles in the reported pattern occurrences. However, precision is just one side of accuracy; the other side being recall, which is the ratio of valid pattern occurrences in the model that actually get detected by the specification. There tends to be a tradeoff between precision and recall. If one over-constrains a specification, precision improves but recall suffers, as valid occurrences may not get reported. Conversely, if one under-constrains a specification, recall improves but precision suffers, as more (valid and invalid) occurrences get detected. The goal is always to find a good compromise between the two metrics.

In this section, we discuss strategies for enhancing recall that a practitioner can apply when specifying design patterns for a given DSML using pQVT. The challenge is the lack of formality of the pattern description. Patterns are often described with informal textual and/or graphical notations. For example, the description of the Adapter pattern in Section 4.2 uses text and a class diagram (Figure 11). This description is still ambiguous. For example, the Object Adapter variant shows a multiplicity of 1 for the anAdaptee attribute, specifying a single instance of Adaptee to delegate to, but one may wonder whether a weaker multiplicity of 0…1 is also valid. Another example is the direct generalization from Adapter to Target shown in the diagram. One may also wonder if an indirect generalization, a weaker constraint, is also valid. A third example is the direct call from the request operation to the realRequest operation. Depending on the intended semantics, one may also consider the weaker constraint of an indirect call.

In order to resolve these ambiguities, a pattern author needs to analyze the semantics of both the specific pattern and the DSML and use the analysis to weaken the constraints of the pattern in a way that still leaves the specification inline with the implied pattern’s intent. As shown previously, constraints in pQVT can take many forms: a specific value for an attribute, a specific association between metaclasses, a relation composition in a when clause or a Boolean expression specified in OCL. The way to weaken an attribute value constraint is to either remove it or to replace it with a weaker Boolean expression. For example, the constraint on the lower bound of the anAdaptee role in Figure 16 (bottom right) can be removed to allow the other valid value of 0, which is still inline with the Object Adapter pattern’s intent of having an instance of Adaptee to delegate to. An alternative would be to replace it with the OCL expression $(\text{lower} = 0 \text{ or } \text{lower} = 1)$ in the when clause.

Similarly, the way to weaken an association constraint is to either remove it, if unnecessary to begin with, or replace it with a transitive version. For example, instead of requiring the Target class to be a direct superclass (an association in the metamodel) of the Adapter class, one can also allow it to be indirect (the Target class is a ancestor of the Adapter class). A common method for defining a transitive association in MOF is to model it as derived with a recursive derivation expression. The expression would calculate the transitive closure of all elements related to a given element using the association. For example, to specify the transitive closure of super
classes of a given class, one can define a new derived allSuperClasses association from metaclass Class to itself with the expression: `self.superClass->union(self.superClass.allSuperClasses)`.

Nevertheless, the above solution would work only if one can change the DSML metamodel to add such a derived association. In a lot of cases, this is not possible since the DSML may be defined by a different organization. One way to get around that in QVTr is to define the derivation directly in the transformation using a query that takes a given element as an argument and returns the closure of related elements as a result. Figure 21 shows the definition of the query along with how it can be called from a relation as a constraint.

```plaintext
relation AdapterPattern {...
  checkonly domain uml Target:Class {}
  checkonly domain uml Adapter:Class {}
  when {
    allSuperClasses(Adapter)->includes(Target);
  }
...}
query allSuperClasses (c : Class) : Set(Class) {
  c.superClass->union(c.superClass->collect(s|allSuperClasses(s)))
}
```

Figure 21 – The Query allSuperClasses

One observation is that the query’s expression is slightly different than the original one because the query is defined in the context of the transformation, not the context of the metaclass Class. Another observation is that the only way to use the query is from `when/where` clauses and not inside domains as other associations in the metamodel, which makes relations a bit harder to read. To address this, we added a small enhancement to QVTr to allow it to define associations in a consistent way with the MOF-defined ones. The idea is based on a similar feature that exists in the QVT Operational language [16], which allows you to define new properties and operations for existing DSML metaclasses that get merged-in in the context of the transformation only. In our case, we allow new derived properties (including association ends) to be defined in the context of existing DSML metaclasses and used in a similar way to metamodel-defined ones within the transformation. The properties will specify their context metaclass, type and derivation expression (specified in the context of the metaclass).

Figure 22 shows an example of how a new derived property allSuperClasses is defined in the context of metaclass Class and then used in the AdapterPattern relation. We also use this property in ClassAdapterPattern to replace the constraint requiring the Adapter class to directly specialize the Adaptee class. We also define a new derived property Class::allOperations (specified in a similar way and thus not shown here) to specify the transitive closure of all operations owned directly or indirectly (inherited) by the class. We then refactor a constraint in AdapterPattern, currently requiring class Adaptee to directly own operation realRequest, by moving it to ClassAdapterPattern only (it is the only constraint for Adaptee in this variant) and use the new transitive property as a constraint in ObjectAdapterPattern. In the latter case, operation realRequest becomes transitively owned by class Adaptee, relaxing the previous constraint it must directly be owned by the type of attribute anAdaptee.

Furthermore, the way to weaken constraints expressed as relation compositions and/or Boolean conditions in `when` clauses is to remove them or replace them with weaker ones. For example, the variant relations for Adapter
(Figure 16 bottom) specify a condition in the when clause that operation request must directly call operation realRequest. The way to relax this condition is to replace it with one that allows for indirect calls, which is more realistic since UML classes often split logic into several operations that call each other. Figure 23 (bottom) shows the new queries localCall and delegationCall that check for both direct and indirect (i.e., after a sequence of local) operation calls. The new queries replace the old ones in the variant relations Figure 23 (top).

```
relation AdapterPattern {...
  checkonly domain uml Target:Class {}
  checkonly domain uml Adapter:Class {
    allSuperClasses = Target:Class {}
  }
  ...
}

property Class::allSuperClasses : Set(Class) =
  self.superClass->union(self.superClass.allSuperClasses);
```

![Figure 22 – The Derived Property allSuperClasses](image)

```
relation ClassAdapterPattern {...
  when {
    localCall(request2, realRequest);
  }
  ...
}

query localCall(caller:Operation, callee:Operation) : Boolean {
  localCall(caller, callee, Set{caller})
}

query localCall(caller:Operation, callee:Operation, stack:Set(Operation)) : Boolean {
  directLocalCall(caller, callee) or
  caller.class.ownedOperation->exists(o| stack->excludes(o) and directLocalCall(caller, o) and
  localCall(o, callee, stack->union(Set{o})))
}

query delegationCall(caller:Operation, link:TypedElement, callee:Operation) : Boolean {
  delegationCall(caller, link, callee, Set{caller})
}

query delegationCall(caller:Operation, link:TypedElement, callee:Operation, stack:Set(Operation)) : Boolean {
  directDelegationCall(caller, link, callee) or
  caller.class.ownedOperation->exists(o| stack->excludes(o) and directLocalCall(caller, o) and
  delegationCall(o, link, callee, stack->union(Set{o}))
}
```

![Figure 23 – The Definition of the localCall and delegationCall Queries](image)

### 4.3.4 Managing Occurrence Type

Up until now we have discussed how to create an accurate specification of a design pattern that detects results that compromise between precision and recall, both of which are affected by the constraints in the specification. Another factor that has an impact on those two metrics is the choice of roles to include in a pattern occurrence or what Pettersson et al. [17] call the occurrence type. In every pattern, there are main roles that characterize it (give the big picture) and others that add details. For example, in the Adapter pattern we find Target, Adapter and Adaptee to be the main roles since they are the most useful in helping a user understand an occurrence of Adapter. Other roles, such as request and realRequest, add detail but can then be traced manually by model inspection. However, if they are reported as well, they may help the user rule out false positives quickly.

On the other hand, reporting non-main roles may harm precision because they are often the ones with most variability (e.g., behavioral information may not exist in UML models to accurately match roles request and realRequest). In order to deal with this situation, we have added a Boolean flag named isMain to the
RoleBinding class of pResults (Figure 10). When this flag is set, it marks those role binding as main. The other roles still get reported but become marked as non-main. Then, when displaying the results, a tool can use that flag to initially hide non-main role bindings from the pResults model tree and give the user the option to show them selectively. We found this method to be highly efficient during results analysis of our case study. Figure 24 shows how we revise the presults domain in the Adapter variant relations to add isMain flags to some role bindings.

```
 enforce domain presults c:Category [name='GoF',
  pattern = p:Pattern [name='Adapter',
    rootBinding = rb1:RoleBinding [role='Target', element=Target:Class{}, isMain=true,
      childBinding = rb2:RoleBinding [role='Adapter', element=Adapter:Class{}, isMain=true,
        childBinding = rb3:RoleBinding [role='Adaptee', element=Adaptee:Class{}, isMain=true,
          childBinding = rb4:RoleBinding [role='request', element=request:Operation{}
        ]
      ]
    ]
  ]
]
```

Figure 24 – The presults Domain with isMain flags

5 Case Study

The pQVT approach has been prototyped using Eclipse-based technologies and has been validated in a case study that involves specifying a sample set of design patterns (a representative subset of GoF patterns) in a sample MOF-based language (UML) and detecting them in a sample model (a design model of a large open-source project). The first objective of the case study is to assess the adequacy of the approach to specify a complex family of patterns with their similarities, differences and variations. The second objective is to assess the accuracy and performance of detection using the approach. The remainder of this section is organized as follows: Section 5.1 discusses the design of the case study including the model and patterns considered; the execution of the case study is covered in Section 5.2; Section 5.3 overviews the tools developed and/or used in the case study; Section 5.4 reports on the results; finally, threats to validity are highlighted along with our mitigations in Section 5.5.

5.1 Case Study Design

Pettersson et al. [17] provides a good comparative framework for pattern detection approaches to facilitate their comparison. The following is a description of how our case study addresses the five axes of that framework:

5.1.1 Design Patterns

The design patterns chosen in the case study are a subset of the GoF design patterns specified for UML. We chose eleven representative patterns from the three known categories: Creational (Abstract Factory, Factory Method and Singleton), Structural (Adapter, Bridge, Composite and Decorator) and Behavioral (Chain of Responsibility, Command, Observer and Strategy). The choices were partly motivated by knowledge of their usage in the example model being analyzed and by their usage in related works to facilitate comparison. We initially specified the original variants mentioned in the GoF book [10] as accurately as we could, balancing both precision and recall metrics, by applying the techniques demonstrated in Section 4.3.
Obviously, it is not plausible to expect to only find patterns that are applications of those original variants. As a matter of fact, we found that for some patterns, different variants were applied in the example model. Those variants were identified manually in the process of defining the gold standard (discussed in Section 5.1.5). Two categories of variants were found: one where some constraints of the original GoF variant were removed (variants of Factory Method, Bridge, and Strategy) and one where some constraints of the original GoF variant were replaced by others (variants of Singleton, Decorator and Observer). An example in the first category was a variant of the Factory Method pattern, where the original constraint requiring class Product to be abstract was removed. An example in the second category was a variant of the Observer pattern, where the original constraint requiring class ConcreteObserver to access the state of class ConcreteSubject through an association was replaced by one allowing the access through a parameter on the update operation. Each of these variants was specified by splitting the original pQVT relation of the pattern into multiple relations: a top relation that specifies the common constraints found in all variants and then one non-top relation for each variant specifying constraints specific to that variant. The former relation composed the latter relations using its where clause, as demonstrated before in Section 4.3.2.

In addition, we found that some GoF relations have common constraints. In these cases, we refactored those constraints into separate top relations and composed them as conditions in the when clauses of those GoF relations. An example of such relation is ObjectRecursion shown in Figure 25 (the names are borrowed from [50]). The relation specifies an abstract class Handler that owns an abstract operation handlerOp. The class is specialized by two distinct concrete classes: Recursor and Terminator, each of which owns a concrete operation that redefines handlerOp. The Recursor class implements its operation to delegate the call to handlerOp on its attribute successor (when clause). Figure 26 shows how the domains of ObjectRecursion bind to the corresponding roles in Composite and Decorator patterns. Notice that the constraints on those roles are identical in both patterns except for the multiplicity of successor, which is * in Composite and 1 in Decorator.

Regarding the behavioral constraints used by GoF, we found that they take one of three general forms: 1) an operation accesses a structural feature (operation/attribute) on the same object; 2) an operation accesses a structural feature on an aggregate object; and 3) an operation creates an object of a given type (mainly for creational patterns). In Section 4.3, we showed how we can define the first two forms. The third form is defined in the same way by detecting objects being created in OCL post-conditions and/or sequence diagrams. In the former, we look for expressions containing oclIsNew operation calls and for the latter we look for messages with messageSort having the value createMessage.
5.1.2 Case Study System

The system that we chose to study is the UML design model of an open source project on Eclipse called the Graphical Modeling Framework (GMF) [43]. GMF is used for building modeling tools on Eclipse and follows closely the familiar Model-View-Controller architecture [11]. The reason we chose to study this system is due to our familiarity with it (used it for years to develop modeling tools) and because it is known to have incorporated GoF design patterns in its design, which makes it an interesting test bed for our approach.

Unfortunately, most public domain projects rarely capture their designs with models and if they do they are usually not released. Since our approach is model-based and focuses on analyzing models (vs. source code) as first level artifacts, we followed a process for recovering the design of GMF as a UML model. The process started with reverse-engineering the structural aspects of the code into UML class diagrams using the facilities of the Rational Software Architect (RSA) tool [44]. The resulting model was then edited manually based on our
knowledge of GMF. First, we added unrecovered details by converting un-typed collections into typed ones, matching attributes to define binary associations, correcting multiplicity and aggregation kind on properties, etc. Then, we transformed the model into a high level design model through a process of manual abstraction including the replacement of dependencies on Eclipse libraries with dependencies on a smaller set of UML interfaces, the replacement of code idioms (e.g., anonymous classes and static constants) with equivalent UML ones (e.g., subclasses and enumerations), and the merging of private operations with other class operations. After that, we added behavioral aspects for operations like OCL post-conditions and interactions (sequence diagrams). Notice that due to the scale of the effort involved, we did not specify behavior for all operations in the model but only for the subset that matched the structural constraints of the pattern roles. We identified this subset by running the detection on the model with structural information only and noting those operations that matched roles that also have behavioral constraints. Finally, the resulting model was reviewed for accuracy by a GMF expert. Table 1 reports some statistics on the final model.

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Attributes</th>
<th>Operations</th>
<th>Interactions</th>
<th>Post-Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>20</td>
<td>42</td>
<td>119</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>Figure</td>
<td>32</td>
<td>38</td>
<td>137</td>
<td>31</td>
<td>52</td>
</tr>
<tr>
<td>Edit</td>
<td>34</td>
<td>39</td>
<td>156</td>
<td>49</td>
<td>96</td>
</tr>
<tr>
<td>UI</td>
<td>9</td>
<td>8</td>
<td>28</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>95</td>
<td>127</td>
<td>440</td>
<td>109</td>
<td>189</td>
</tr>
</tbody>
</table>

### 5.1.3 Accuracy Metrics

When analyzing the results of pattern detection, a pattern occurrence can be classified into one of four categories: true-positive (TP: correctly found), false-positive (FP: incorrect found), true-negative (TN: incorrectly unfound) and false-negative (FN: correctly unfound). Two common metrics of measuring the accuracy of detection results are then precision and recall [17]. Precision is the ratio of correctly found to all found occurrences and equals TP / (TP + FP). Recall is the ratio of correctly found to all correct occurrences (gold standard) and equals TP / (TP + FN). A perfect precision score of 1 means that every occurrence found was correct (but says nothing about whether all correct occurrences were found), whereas a perfect recall score of 1 means that all correct occurrences were found (but says nothing about how many incorrect occurrences were also found). Both measures are therefore complementary.

### 5.1.4 Occurrence Type

A pattern occurrence type is the set of role bindings that make up a unique occurrence of a pattern. In pQVT, a pattern occurrence is represented (in pResults) as a compact tree of role bindings. Recall from Section 4.3.4 that role bindings marked as main (isMain=true) play a significant role in helping a user quickly analyze a detected pattern occurrence without knowing all the details. In our case study, we define the occurrence type for each pattern we study, mark a subset of role bindings as main and then calculate the metrics (precision and recall) for
two separate cases: (A) with all role bindings and (M) with main role bindings. The first case reports on the accuracy of a full analysis of results, while the second reports on the accuracy of a partial analysis (main roles only). The goal is to discover when it is useful to perform a partial analysis since it is faster (less roles to analyze).

Our criterion for choosing main roles in this case study includes roles that are played by classes that are the most abstract in their inheritance hierarchy or that are related to such classes through associations. For example, in the Adapter pattern (Figure 11), the Target and Adaptee classes are both the most abstract in their inheritance hierarchy, while the Adapter class is related to the Adaptee with an association (in the Object Adapter variant). This results in the following selection of main roles: Singleton (Singleton), Abstract Factory (AbstractFactory, AbstractProduct), Factory Method (AbstractFactory, AbstractProduct), Adapter (Target, Adapter, Adaptee), Bridge (Abstraction, Implementation), Composite (Component, Composite), Decorator (Component, Decorator), Chain of Responsibility (Handler), Command (Command, ConcreteCommand, Receiver), Observer (Observer, Subject) and Strategy (Context, Strategy).

5.1.5 Gold Standard

A gold standard is the set of actual occurrences of design patterns in the system and is used for assessing the accuracy of the detection. In our case study, the pattern occurrences in the gold standard were discovered by two methods: a) some were known from our previous experience with the system; b) some were detected by relaxing several behavioral and structural constraints in the pattern specifications then manually verifying the result against the intent of the patterns. As shown in column (G) of Table 3, the total number of unique occurrences of all patterns in the gold standard is 148 in the (A) case (considering all roles) and 45 in the (M) case (considering main roles). The column also shows the number of unique occurrences per pattern in the same two cases.

5.2 Case Study Execution

UML models are defined for many purposes ranging from informal communication of design ideas all the way to being used in model-driven engineering to drive system implementations. As such, UML models can have different levels of details. One of the objectives of the case study is to investigate the impact of varying levels of model details on the accuracy of detecting GoF patterns. Another objective is to evaluate the impact of having unknown pattern variants on the detection results and whether it is worth identifying and specifying them. A third objective is to assess the impact on accuracy when considering all the roles vs. only the main roles of a pattern.

To satisfy these three objectives, the case study is structured into a matrix of experiments with three axes of variability: 1) model details, 2) specified variants and 3) considered roles. In the first axis we vary the level of details in the design model. The kinds of details we consider include: structural details (class diagram) (S), interactions (sequence diagrams) (I) and OCL post-conditions (C). These details are then combined into four separate sub-experiments: S, S+C, S+I and S+I+C. In the second axis, we consider different pattern variants. In
the first sub-experiment, we consider the Original GoF variants only (shown in Table 3). In other sub-experiments, we add variants that are actually used in the GMF model. In order to ensure prior knowledge of those variants does not unfairly skew the results, we take a pragmatic approach of manually inspecting a small part (one package) of the model looking for variants then adding those variants when detecting occurrences in the whole model. Since the model is divided into four packages (Table 1), we define four other sub-experiments (named after the package): Model variants, Figure variants, Edit variants and UI variants. In the third axis, we have two sub-experiments: one where all the role bindings are considered (A) and one where only the main roles are considered (M). In each one of those sub-experiments in the matrix, we collect three numbers: the total number of detected occurrences (T), the precision ratio of the result (P) and the recall ratio of the result (R). The last two are calculated based on comparing detection results with the gold standard (also defined for the two cases A and M). When the total number of occurrences is 0 the ratios are reported as undefined (U).

5.3 Tooling

The approach and the case study have been prototyped using a number of Eclipse-based technologies. For example, the UML design model was defined using the RSA modeling tool [44]. RSA uses the UML2 [45] open-source implementation of the UML 2.2 specification on Eclipse. UML2 is defined using the Eclipse Modeling Framework (EMF) [46], an Eclipse implementation of a subset of MOF. We used EMF to define the pResults language and developed a tree-based viewer for its models (Figure 27). For each element in the pResults model, we roll up a pair of numbers \((x, y)\) where \(x\) represents the number of unique pattern occurrences below that element (in the tree) considering all roles, and \(y\) is the same except considering only the main roles. Those numbers can help a user selectively drill down the results.

For the QVTr transformation, we used the open-source tool Medini QVT [47] after improving its performance. In particular, we improved the algorithm that binds variables of a relation to elements from the candidate models that satisfy the relation’s constraints. The old algorithm builds a search tree for candidate model elements using the variables’ inter-relationships. Each branch from root to leaf represents a unique complete binding of variables to candidate elements that gets checked using constraints at the leaves. In the new algorithm, constraints are checked as soon as their affected variables are bound, allowing the tree branches to be pruned much earlier and therefore cutting down the search space faster. The other improvement we made relates to the execution time of reusable queries. Instead of repeatedly executing the query on the same parameter values, we cache the results on the first call and reuse them on subsequent calls.

The time needed to run any of the pattern detection experiments in the matrix was in the range of 15-80 seconds (Table 2) on a laptop with 2.4 GHz core 2 duo processor and 3G of memory. Results show that when the model had structural information only (S), detection was the slowest because behavioral constraints were not effective at pruning the search space. When behavioral information was added detection speed improved...
dramatically due to the effective pruning of the search space by behavioral constraints. Having interactions only (S+I) was faster than having post-conditions only (S+C) since the model had much less interactions than post-conditions (Table 1) for the specifications to process. When other variant got also considered, detection slowed down an average of 21% due to checking alternative constraints. Considering that the model was not small (Table 1), these results suggest that pattern detection with pQVT is fast enough to be run repeatedly by users trying to analyze the impact of their model changes on patterns.

| Table 2 – Time in Seconds for Running the Experiments in the Matrix |
|-------------------------|----------------|----------------|----------------|
| Experiment              | S             | S+C            | S+I            | S+C+I          |
| Original GoF Variants   | 52            | 23             | 15             | 30             |
| + Other GoF Variants    | 65            | 27             | 18.5           | 36             |

**5.4 Results**

In this section, we discuss the accuracy results of the pattern detection experiments quantitatively and provide qualitative insights to explain them. In Section 5.4.1, we discuss the first experiment in the matrix that involves structural details only, original GoF variants, and considering all roles. In Section 5.4.2, we discuss the impact of adding behavioral details to the model on accuracy. In Section 5.4.3, we discuss the impact of specifying other variants on accuracy. In Section 5.4.4, we discuss the impact of considering only the main roles on accuracy. Finally, in Section 5.4.5 we summarize the results of the case study and discuss their practical implications.

**5.4.1 First Experiment**

In this experiment, we detect the original GoF variants in the GMF model having structural details only (column S in Table 3) and calculate the accuracy using all roles (A). The general observation is that the result has very low precision (average of 0.029) but good recall (average of 0.851). One reason for the low precision is that several GoF patterns do not have enough structural constraints to be unambiguously recognized, like Bridge (P=0.008) and Strategy (P=0.017). Another reason is that most GoF patterns, like Adapter (P=0.041), have operation roles that are hard to detect accurately without their behavioral information. Some exceptions exist like Singleton (P=1.0), where the accessor operation is structurally unique (required to be static and have its type equal to its owner) and Composite (P=0.9), where the pattern has enough structural constraints (e.g., attribute children has a multiplicity of *, is owned by a class and is typed by its super class), reducing chances for false positives.
Conversely, recall (R) is good because in the absence of any behavioral information, behavioral constraints such as localCall and delegationCall become automatically satisfied (recall from Section 4.3.2 that when neither a post condition nor a sequence diagram is available, the constraint checking behavior returns true). Some exceptions exist, like Observer (R=0.25) and Decorator (R=0.4), where patterns have variants (other than the original ones) used in the model but not specified and included in the detection.

Table 3 – Results of the Original GoF Variants Experiment

<table>
<thead>
<tr>
<th>Pattern (A/M)</th>
<th>G</th>
<th>S</th>
<th>S+C</th>
<th>S+I</th>
<th>S+C+I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>P</td>
<td>R</td>
<td>T</td>
<td>P</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>A</td>
<td>6</td>
<td>78</td>
<td>0.077</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>2</td>
<td>6</td>
<td>0.333</td>
<td>1.000</td>
</tr>
<tr>
<td>Factory Method</td>
<td>A</td>
<td>8</td>
<td>156</td>
<td>0.045</td>
<td>0.875</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>3</td>
<td>7</td>
<td>0.286</td>
<td>0.667</td>
</tr>
<tr>
<td>Singleton</td>
<td>A</td>
<td>4</td>
<td>2</td>
<td>1.000</td>
<td>0.500</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>4</td>
<td>2</td>
<td>1.000</td>
<td>0.500</td>
</tr>
<tr>
<td>Adapter</td>
<td>A</td>
<td>33</td>
<td>805</td>
<td>0.041</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>9</td>
<td>33</td>
<td>0.273</td>
<td>1.000</td>
</tr>
<tr>
<td>Bridge</td>
<td>A</td>
<td>17</td>
<td>1420</td>
<td>0.008</td>
<td>0.706</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>5</td>
<td>23</td>
<td>0.174</td>
<td>0.800</td>
</tr>
<tr>
<td>Composite</td>
<td>A</td>
<td>18</td>
<td>20</td>
<td>0.900</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>2</td>
<td>3</td>
<td>0.667</td>
<td>1.000</td>
</tr>
<tr>
<td>Decorator</td>
<td>A</td>
<td>5</td>
<td>6</td>
<td>0.333</td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>2</td>
<td>2</td>
<td>0.500</td>
<td>0.500</td>
</tr>
<tr>
<td>Chain Of Resp.</td>
<td>A</td>
<td>4</td>
<td>6</td>
<td>0.667</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>1</td>
<td>2</td>
<td>0.500</td>
<td>1.000</td>
</tr>
<tr>
<td>Command</td>
<td>A</td>
<td>23</td>
<td>731</td>
<td>0.031</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>6</td>
<td>27</td>
<td>0.222</td>
<td>1.000</td>
</tr>
<tr>
<td>Observer</td>
<td>A</td>
<td>8</td>
<td>150</td>
<td>0.013</td>
<td>0.250</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>5</td>
<td>1</td>
<td>1.000</td>
<td>0.200</td>
</tr>
<tr>
<td>Strategy</td>
<td>A</td>
<td>22</td>
<td>1025</td>
<td>0.017</td>
<td>0.773</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>6</td>
<td>15</td>
<td>0.333</td>
<td>0.833</td>
</tr>
<tr>
<td>Overall</td>
<td>A</td>
<td>148</td>
<td>4399</td>
<td>0.029</td>
<td>0.851</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>45</td>
<td>121</td>
<td>0.289</td>
<td>0.778</td>
</tr>
</tbody>
</table>

5.4.2 Adding Behavioral Details

In these experiments, we assess the impact of adding behavioral details (to the model) on accuracy. When we only add OCL post-conditions (column S+C in Table 3), we observe that the precision of few patterns, like Singleton and Adapter, jumps to the perfect score (P=1.0). While we believe that these perfect results are partly due to chance, it generally indicates that having post-conditions can reduce false positives since specifications can use them to check behavioral constraints like operation calls and object instantiations. However, we also observe a dramatic decrease in the total number of detected occurrences (from 4399 to 6). In fact, the majority of patterns do not get any detected occurrences leaving their precision undefined. The reason is that expressions in OCL, a side-effect free constraint language, cannot have their AST containing CallOperationExp nodes (Section 4.3.2) referencing non-query operations (i.e., those with side-effects). This means that behavioral constraints (e.g., localCall and delegationCall) checking OCL post-conditions for calls to non-query operations cannot be
satisfied. Indeed, patterns having this kind of operation playing a role (e.g., operation update in Observer that updates the observer’s state) are not detectable in this case causing recall to fall dramatically (average is 0.041).

On the other hand, when we only add interactions (sequence diagrams) only to the model (column S+I in Table 3), we observe a radical improvement in precision (average is 0.826) across the board. In fact, more than half the patterns get a perfect precision score (P=1.0), especially ones classified as behavioral (and some structural too). The reason goes back to interactions being much more expressive than OCL allowing for specifying a wider range of behavioral information (including calls to non-query operations) thus increasing the chance of satisfying behavioral constraints. However, the reason some detected occurrences are still invalid is because their pattern specifications do not have enough constraints to disambiguate them (e.g., Adapter and Command). One exception to the high precision is Singleton, where the precision becomes undefined as no occurrences get detected. This is because the constraint requiring a Singleton class to have a single instance is expressed in OCL, which is not analyzed in this experiment. Finally, we observe that recall remains almost unchanged (average 0.831) since most valid occurrences that matched structurally before also matched behaviorally this time. This indicates that interactions alone are very effective in expressing the behavioral details of the model.

In the third experiment we add both OCL post-conditions and interactions to the model (column S+C+I in Table 3). We observe that precision (average is 0.829) and recall (average is 0.851) only slightly improve compared to the interactions only (S+I) experiment. This is because post-conditions only contribute a small number of occurrences and some of those even overlap with those contributed by interactions only. This suggests that adding post-conditions to models with interactions may not be worth the extra effort to improve accuracy.

### 5.4.3 Specifying Other Variants

In these experiments, we assess the impact of detecting additional pattern variants on accuracy. As discussed in Section 5.2, we simulate discovering variants by manually inspecting one package of the model in each additional experiment (named after the package). We then attempt to detect the pattern variants on all packages. The patterns that have variants discovered by manual inspection in each package are: **Model** (Singleton, Observer), **Figure** (Singleton, Decorator and Observer), **Edit** (Bridge, Observer and Strategy) and **UI** (Factory Method).

<table>
<thead>
<tr>
<th>Experiment (A/M)</th>
<th>G</th>
<th>S</th>
<th>S+C</th>
<th>S+I</th>
<th>S+C+I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>P</td>
<td>R</td>
<td>T</td>
<td>P</td>
</tr>
<tr>
<td>Model Variants</td>
<td>A</td>
<td>148</td>
<td>6405</td>
<td>0.021</td>
<td>0.905</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>45</td>
<td>127</td>
<td>0.323</td>
<td>0.911</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>148</td>
<td>6408</td>
<td>0.021</td>
<td>0.926</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>45</td>
<td>128</td>
<td>0.328</td>
<td>0.933</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>148</td>
<td>7689</td>
<td>0.018</td>
<td>0.959</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>45</td>
<td>127</td>
<td>0.323</td>
<td>0.911</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>148</td>
<td>4407</td>
<td>0.029</td>
<td>0.858</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>45</td>
<td>122</td>
<td>0.295</td>
<td>0.800</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>148</td>
<td>6227</td>
<td>0.022</td>
<td>0.912</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>5</td>
<td>126</td>
<td>0.317</td>
<td>0.889</td>
</tr>
<tr>
<td>Overall</td>
<td>A</td>
<td>148</td>
<td>6227</td>
<td>0.022</td>
<td>0.912</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>5</td>
<td>126</td>
<td>0.317</td>
<td>0.889</td>
</tr>
</tbody>
</table>
The detection results for each experiment are shown in Table 4 in an aggregated form (i.e., for all patterns together). We observe that average precision slightly improves across the board due to the fact that the additional variants are specified based on their actual application in the model and therefore are more precise. One exception where precision slightly dips is when detecting with structural information only. The reason is that the new variants add precision thanks to their behavioral constraints and therefore in the absence of behavioral information, these constraints do not help eliminate new false positives. We also observe that average recall improves across the board. This is because more valid occurrences become detectable with the new variants.

5.4.4 Considering Main Roles

In these experiments, we assess the impact of considering the main roles only (vs. all roles) on accuracy. Recall that the subset of main roles can be used for a quick inspection of the results to get the big picture before diving into the details (Section 5.1.4). We observe that average precision improves dramatically (10 fold) in the case of models with structural information only, although it is still relatively low (approximately 0.3). The reason is that our criteria for selecting main roles lead to selecting class roles only; operation roles that are hard to detect accurately with structural information only (and harmed precision before) are not considered now. On the other hand, we observe that average precision in the cases where models have behavioral information dips (15%) although it is still relatively high (approximately 0.7). The reason is that roles that benefit the most from the extra precision (resulting from the behavioral information) are the operation roles, which are not chosen as main. We also observe that the average recall dips when the main roles are considered only. This indicates that the subset chosen as main roles (i.e. the classes) has stronger constraints relative to the whole set of roles. We also observe that recall dips more when the original variants are detected only (10%) than when the other variants are included (4%) because accounting for variants weakens the overall constraints leading to more occurrences matching.

5.4.5 Summary of Results

The case study results suggest that pQVT is an adequate approach for specifying a complex family of design patterns, such as GoF, at the language (metamodel) level. With its reuse semantics, including relation composition and queries, the approach can incrementally refine pattern specifications with increasing levels of details and variants in order to achieve acceptable and balanced recall and precision.

In the specific case of GoF, the case study indicates that when UML models have structural information only, pattern detection results have low precision (3%) but acceptable recall (85%). When interactions alone are added to the model, precision improves dramatically (83%), while recall remains almost unchanged. When OCL post-conditions alone are added, precision improves for some patterns at the expense of both the number of true positives and recall. When adding both kinds of behavioral information, precision improves only slightly over the interactions case. This suggests that while OCL post-conditions may be useful in formalizing behavior of
operations, their impact on the accuracy of pattern detection is generally low. Furthermore, when variants are manually inspected in part of the model and used to detect occurrences in the whole model, recall improves further (90%) though the change in precision (84%) is only limited. This suggests that in practice, companies should identify and specify pattern variants on their projects and use them to detect patterns on future projects. Finally, results suggest that users should consider analyzing detected occurrences by inspecting the main role bindings first for added precision, especially when their models have structural information only. They can then inspect other role bindings as needed to gain more insight.

5.5 Threats To Validity

One threat to validity of the case study is the choice of domain being UML and not a specific DSML. However, we have demonstrated that all the specifications are done at the language level using the language’s specific MOF-based metamodel and as such we are confident the approach can easily apply to specify patterns of any MOF-based DSML. Another threat is the choice of patterns and whether they are diverse and complex enough. To mitigate this threat, we have specified 11 patterns (out of 23) from the three different categories (creational, structural and behavioral) of GoF, some of which have many constraints and several variants (Section 5.1.1). Another threat is the choice of system to study and whether it is big enough and realistic. To mitigate this risk, we have chosen a large open-source system that is known to have used GoF design patterns. We have modeled its design using a rigorous process (reverse-engineering with a tool, followed by manual abstraction, then a review by an expert) and with varying levels of details such as class diagrams, sequence diagrams and post-conditions (Section 5.1.2). Another threat to validity is the construction of the gold standard since it is the basis for calculating accuracy. To mitigate this risk, we used two methods to find the valid occurrences of the known GoF variants: our previous knowledge of the system from past experience and a semi-automatic process of relaxing several constraints in the pattern specifications followed by manually verifying the detection results to identify true positives (Section 5.1.5).

6 Limitations and Future Work

We identify two known limitations in the approach presented in this paper that we plan to address in future work. We also highlight other directions to take this work forward.

6.1 Pattern Specification Genericity

In some cases, the same pattern needs to be specified on different but related metaclasses in the same language. For example, the Composite pattern may need to be defined on both classes and data types in UML. One way to address this requirement is to specify a pattern in terms of more abstract elements in the metamodel. For example, metaclasses Class and DataType are related through their common abstract super class Classifier (Figure
As shown previously (Figure 25), most of the constraints of the Composite pattern can be extracted into a reusable relation called ObjectRecursion. That version of the relation uses the metaclass Class as a type for the domain variables Handler, Recursor and Terminator. We show another version of the relation in Figure 29 where the abstract metaclass Classifier is used as a type for the variables instead. In addition, the new version uses more general metaclasses to relate those variables. MOF has two mechanisms for defining property specialization: the first is property subsetting (i.e., declaring a property as a subset of another); and the second is property redefinition (i.e., declaring a property as a replacement for another inherited one). In Figure 28, we see that property ownedOperation in both metaclasses Class and DataType is annotated {subsets feature}, which means it is a subset of property feature inherited from metaclass Classifier. This makes the contents of ownedOperation also available through feature and hence feature can act as a replacement for ownedOperation in the relation. Similarly, feature can also act as a replacement for property ownedAttribute. The other part of the relation to generalize is the derived property allSuperClasses (defined in Figure 21), which is only applicable to classes. We define a more general form of this property, called allGenerals, by replacing the property superClass by the property general from Classifier (notice that superClass is annotated with {redefines general} in the metamodel meaning it is a special case of general). Both properties are shown in Figure 28. The new version of the relation ObjectRecursion is now ready to be composed by the Composite pattern defined on DataType, as shown in Figure 30.

On the other hand, if the same pattern needs to be defined on unrelated metaclasses in the same metamodel or ones that come from different metamodels then the approach does not currently have a mechanism to concisely specify that. We call such mechanism: pattern specification genericity, i.e., the ability to specify a pattern generically and then bind it to different metaclasses (from the same or different metamodels) to produce different compatible instances of the same specification. While this mechanism is out of the scope in this paper, we plan to address it in a future work. One way to do it is to add generics support to the QVTr language to allow it to define generic transformations that can be instantiated to produce concrete ones. This is similar to the generics support in programming languages like Java (although this came late in version 5).
6.2 Partial Pattern Matching

In Section 4.3.2, we showed how we can enhance the recall of pattern detection by specifying known variants of the patterns that share common constraints. Also in Section 5.4.3, we showed how this technique was applied to the GoF patterns to enhance recall to around 90%. However, while this takes care of anticipated pattern variations in the model, it does not currently address unanticipated variations. In other words, the approach can report only on exact matches of patterns but not partial matches. One way to address this limitation is to allow some constraints to be defined as optional and report on their dissatisfaction without invalidating an occurrence. The reporting can assign scores to those constraints and calculate a confidence ratio based on how many of them are satisfied. While the impact on recall is clear in this case, we would need to study the impact on precision as well. We leave this to future work.

6.3 Other Directions

One possible direction could be to apply pQVT to patterns of profile-based DSMLs (e.g., SysML). Unlike metamodels, which are defined as instances of MOF, profiles are defined as instances of UML. Patterns defined
for a profile have their roles played by stereotypes from a profile vs. metaclasses from a metamodel. Therefore, it would be interesting to show how such patterns can be specified using pQVT (e.g., how the roles get constrained by stereotypes as types and how stereotype properties are constrained by values). Another possible direction is to show how pQVT can be used to specify patterns of other DSMLs (e.g., the control flow patterns of BPMN) or cross-DSML patterns (e.g. involving diagram interchange models and their corresponding abstract syntax models). Another possibility is to use pQVT to specify domain-specific anti-patterns (design pitfalls).

7 Conclusion

Detecting occurrences of known design patterns supports the activities of model comprehension and maintenance.

We presented pQVT, a novel approach to detecting design patterns in MOF-based models. The approach can be used to specify design patterns of any MOF-based DSML at the language level, i.e., using the abstract syntax of the DSML. This allows for building generic detection tools that work for any DSML and reduces the learning curve for users detecting patterns in multiple DSMLs. pQVT defines patterns with a QVTr transformation from a DSML model, in which elements playing pattern roles are identified, to a newly defined DSML model for reporting identified pattern occurrences called pResults. Using pQVT, patterns can be specified in a precise way by encoding various kinds of constraints. Patterns variability is addressed by specifying multiple known variants of the pattern, which allows for recovering most of the pattern occurrences. The approach provides both a textual and a graphical notation and defines reuse semantics, including relation composition and reusable queries, which can be used to define reusable specification fragments that can be combined in different contexts, allowing patterns to be specified in a modular way. It can also be used to incrementally refactor pattern specification to accommodate variants as they are discovered.

We also reported on a case study where the approach was prototyped and used to specify 11 (out of 23) patterns in the three categories of the popular GoF family of patterns, some of which are complex and have several variants. The specifications were then used to detect patterns in a design model of a large open source system that had 148 unique occurrences of those GoF patterns. The case study showed that pQVT is adequate to specify patterns in a concise and modular way and yields good levels of precision and recall when the model is at an adequate level of detail. More specifically, our quantitative and qualitative analyses of the detection results suggest that GoF patterns can accurately be detected (with precision of 84% and recall of 90%) in UML design models when class diagrams and behavioral diagrams (e.g., sequence diagrams) exist in the model and when known variants are considered. On the other hand, the specification of post-conditions (in OCL) does not result in significant added benefits to accuracy. Therefore, it is recommended for organizations to capture their use of pattern variants in past projects to help the accuracy of detection in future projects.
References


[6] Diagram Definition (DD) Revised Submission v.91. ad/10-05-01


[16] Query/View/Transformation (QVT) v1.0. formal/2008-04-03


